Security assessments for software that matters.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Pentest-Report IVPN Unlinked Service Access Feature 07.2026

Cure53, Dr.-Ing. M. Heiderich, Dr. D. Bleichenbacher, Dr. N. Kobeissi

Index

- Introduction
- Scope
- Identified Vulnerabilities
  - IVP-09-001 WP4: Shared HMAC key enables manifest signature forgery (High)
  - IVP-09-002 WP4: Subscription revocation not propagated to client store (Medium)
  - IVP-09-004 WP4: Stale manifest replay accepted within validity window (Low)
  - IVP-09-007 WP3: SQL injection via manifest-derived tier in MySQL path (Medium)
  - IVP-09-008 WP3: Unbounded gzip decompression enables DoS (Medium)
- Miscellaneous Issues
  - IVP-09-003 WP4: SHA-512 pre-hash reversible by HSM provider (Low)
  - IVP-09-005 WP4: Single dual-purpose key lacks rotation & versioning (Low)
  - IVP-09-006 WP1: Absent rate limiting enables HSM DoS (Low)
  - IVP-09-009 WP4: Cryptography disabled and non-safeguarded by default (High)
  - IVP-09-010 WP3: HSM gRPC lacks mandatory mTLS authorization (Medium)
  - IVP-09-011 WP3: Manifest fetch lacks TLS or pin distributor enforcement (Medium)
  - IVP-09-012 WP3: Lenient token parsing (Low)
- Conclusions

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Introduction

“What you do online can be tracked by organizations you may not know or trust and become part of a permanent record. A VPN can’t solve this on its own, but can prevent your ISP from being able to share or sell your data.”

From https://www.ivpn.net/en/

This document verifies the results of a penetration test and source code audit against the IVPN Unlinked Service Access feature, which was performed by Cure53 in Q3 2026.

The engagement was commissioned by IVPN Limited in May 2026 to appraise the product’s security posture and offer actionable remediation advice for all detected vulnerabilities. Three cybersecurity experts conducted the investigations in CW28 July 2026 with a resource allocation of ten days, which was deemed sufficient to achieve the necessary degree of coverage.

Four separate work packages (WPs) were devised when compiling the scope in question. These were given the following headings:

- **WP1:** White-box pen.-tests & audits against IVPN token & pre-auth services
- **WP2:** White-box pen.-tests & code audits against IVPN distribution services
- **WP3:** White-box pen.-tests & code audits against IVPN verifier & clients
- **WP4:** White-box pen.-tests & code audits against IVPN crypto & privacy model

The internal maintainers granted unfettered access to all assets needed to facilitate the white-box approach, such as sources and similar. The items also helped with the preparation measures; finalized in late June and early July 2026 (CW27), these tasks generally served for test optimization and expedition purposes.

Communications throughout the procedures were handled using a dedicated and shared Rocket.Chat channel between the teams of IVPN and Cure53. All team members that actively contributed to this pentest were invited to join the channel in advance.

The collaboration significantly enhanced the engagement’s efficacy. Furthermore, no significant roadblocks were encountered, owing to the meticulous initial preliminary actions and the ongoing support provided by the development team. Hindrances and delays were wholly avoided.

Live reporting was offered and implemented via the aforementioned Rocket.Chat forum. To accompany this, Cure53 relayed a number of progress updates to the development team.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Cure53 believes that a good degree of analysis was achieved over the WP1-WP4 attributes. Twelve findings were confirmed and documented in ticket format via two dedicated categories, *Identified Vulnerabilities* and *Miscellaneous Issues*. Five of the former and seven of the latter were located.

This audit of the IVPN Unlinked Service Access feature yielded a relatively favorable impression. The cryptographic foundation is built on state-of-the-art primitives that are naturally resilient to misuse, while the privacy-preserving unlinked architecture itself is thoughtfully and robustly designed.

Nevertheless, Cure53 identified certain elements that would benefit from further improvements. On a positive note, none of the identified issues stem from flawed protocol constructions, which is a reassuring testament to the evidently secure engineering principles administered by the IVPN team.

The vulnerabilities generating the greatest concern arise from the reuse of the same cryptographic keys across entirely different security contexts. Moreover, the stack is shipped with the strongest transport protections and security controls disabled in the default test mode. Secondary operational weaknesses such as SQL injection risks, a lack of rate limiting, and malleable token parsing were also observed. Conclusively, the platform is highly capable of operating securely, yet is never actually forced to do so by its own codebase.

All in all, Cure53 is pleased to note that safeguarding the inspected feature for production does not require cryptographic redesigns. Albeit, the internal team should focus on implementing strict key isolation and imposing secure, hardened configurations by default.

For ease of navigation, this report is structured into several key sections. It starts with the *Scope* overview, which presents the core testing parameters and setup information in a concise bulleted list. Next, the identified security flaws are logged chronologically as they were discovered, split into two categories: *Identified Vulnerabilities* and *Miscellaneous Issues*. Each entry provides a detailed technical breakdown, a Proof-of-Concept (PoC) or reproduction steps, relevant code snippets, and specific remediation advice. The document concludes with a summary of Cure53's overall assessment of the target systems, based on the evidence gathered.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Scope

- **Pen.-tests & code audits against IVPN's Unlinked Service Access feature**
  - **WP1:** White-box pen.-tests & audits against IVPN token & pre-auth services
    - **Source code:**
      - https://github.com/ivpn/unlinked-access/tree/main/services/token
      - https://github.com/ivpn/unlinked-access/tree/main/services/preauth
    - **Commit hash:**
      - 8162aaf1e341f671f326f13dca14183bee581299
  - **WP2:** White-box pen.-tests & code audits against IVPN distribution services
    - **Source code:**
      - https://github.com/ivpn/unlinked-access/tree/main/services/distributor
    - **Commit hash:**
      - 8162aaf1e341f671f326f13dca14183bee581299
  - **WP3:** White-box pen.-tests & code audits against IVPN verifier & clients
    - **Source code:**
      - https://github.com/ivpn/unlinked-access/tree/main/services/verifier
    - **Commit hash:**
      - 8162aaf1e341f671f326f13dca14183bee581299
  - **WP4:** White-box pen.-tests & code audits against IVPN crypto & privacy model
    - **Source code:**
      - https://github.com/ivpn/unlinked-access/tree/main/services/token
      - https://github.com/ivpn/unlinked-access/tree/main/services/verifier
    - **Commit hash:**
      - 8162aaf1e341f671f326f13dca14183bee581299
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following section lists all vulnerabilities and implementation issues identified during the testing period. Notably, findings are cited in chronological order rather than by degree of impact, with the severity rank offered in brackets following the title heading for each vulnerability. Furthermore, all tickets are given a unique identifier (e.g., *IVP-09-001*) to facilitate any future follow-up correspondence.

IVP-09-001 WP4: Shared HMAC key enables manifest signature forgery *(High)*

Cure53 observed that the token service exposes a single gRPC method, Generate, which returns *base64(HMAC-SHA256(SHA-512(input)))* for any input up to 4096 bytes. However, the implementation fails to verify the input's format. The manifest generator signs each manifest by calling the same method with the manifest digest as its input; as such, manifest signatures are identical to tokens with *account_id* as the digest. The verifier accepts any manifest for which *MacVerify(SHA-512(digest), signature)* holds.

Because per-account token generation and manifest signing share an HSM key and unlabeled operation, any party able to invoke Generate on a chosen input can produce a valid manifest signature without holding the HSM MAC-generate credential. The oracle is reachable via two methods. Firstly, a holder of a token-service mTLS client certificate (held by both the generator and the Internet-exposed preauth service) can call Generate directly. Secondly, a holder of *PREAUTH_ADD_PSK* can submit the digest as *account_id* to the preauth */v1/preauth/add* endpoint, which delivers the resulting raw token to the configured *SESSION_URL* webhook.

An attacker in possession of a forged signature and able to place the manifest in the location of the verifier's fetch (i.e., the shared manifest volume, distributor, or a network position on MANIFEST_URL) can rewrite the *is_active*, *active_until*, and tier fields of existing MailX subscriptions. This activity grants or withdraws access in an arbitrary manner, as well as enables the configuration of a perpetual attacker-selected *valid_until*. A generator compromise yields both the signing oracle and the delivery path in a single step, breaking the guarantee that only the HSM key holder can author manifests.

Affected file #1:
services/token/service/service.go

Affected code #1:
```
func (s *Server) Generate(ctx context.Context, req *proto.Request)
(*proto.Response, error) {
    if len(req.Input) > maxInputBytes {
        return nil, fmt.Errorf("input exceeds maximum allowed size of %d
bytes", maxInputBytes)
    }
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
[...]
token, err := s.generateToken(reqCtx, req.Input)
```

Affected file #2:
services/token/service/service.go

Affected code #2:

```
digest := sha256.Sum256(data)
digestBase64 := base64.StdEncoding.EncodeToString(digest[:])
signature, err := s.Token.GenerateToken(digestBase64)
```

Steps to reproduce:

1. Build a manifest and compute its digest *D = base64(SHA-256(compact JSON with the signature field unset))*.
2. Submit D as the Generate input using a token-service client certificate or as *account_id* to preauth */v1/preauth/add*.
3. Read the returned token from the *SESSION_URL* webhook.
4. Set the manifest signature to this token and serve the manifest to the verifier.
5. Observe that verification succeeds.

To mitigate this vulnerability, Cure53 advises leveraging two distinct HSM keys, one for token generation and another for manifest signing. This will ensure that a token-generation credential cannot authorize manifest signatures. Alternatively, the dev team should apply explicit domain separation by prefixing a fixed context label to the pre-hash input on both the signing and verification implementations, e.g., *ivpn-token* and *ivpn-manifest*. Lastly, the *i-XXXX-XXXX-XXXX* account ID grammar should be enforced on both the Generate input and preauth *account_id* field to guarantee that manifest digests can never be submitted as tokens.

IVP-09-002 WP4: Subscription revocation not propagated to client store *(Medium)*

Cure53 noticed that the verifier applies a manifest by iterating the rows present in each MailX store and updating only those with *token_hash* in the manifest. Rows absent from the manifest remain untouched. The generator, in turn, only includes an account while it holds an active service that has been active within the previous 14 days. As a result, when an account is cancelled out of band (for fraud, a refund, a chargeback, or a policy violation), it drops out of the next manifest entirely, rather than appearing with *is_active* set to *false*. Since the verifier never receives a row that deactivates it, the corresponding MailX subscription retains its last-published state.

In light of this, a cancelled IVPN subscriber will continue to be treated as active by MailX until the previously-published *active_until* date elapses, which is rounded up to noon UTC of the expiration day and can be multiple weeks in the future.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Immediate revocation does not propagate, while correct behavior depends entirely on MailX enforcing *active_until* on its own. In the common schema whereby an active service implies an active account, the manifest never carries *is_active* false, meaning that the deactivation path effectively remains unused.

Affected file:
services/verifier/service/service.go

Affected code:

```
func UpdateSubscriptionFromManifest(sub model.Subscription, manifestSubs
[]model.Subscription) (model.Subscription, error) {
    for _, s := range manifestSubs {
        if sub.TokenHash == s.TokenHash {
            sub.IsActive = s.IsActive
            sub.ActiveUntil = s.ActiveUntil
            sub.Tier = s.Tier
            return sub, nil
        }
    }
    return model.Subscription{}, fmt.Errorf("subscription %s not found in
manifest", sub.ID)
}
```

To mitigate this vulnerability, Cure53 advises introducing an explicit signed deactivation mechanism, such as a tombstone entry or a signed revocation list, so that revocation is affirmative rather than inferred from a manifest omission. The implementation should distinguish genuine revocations from transient outages in which accounts are temporarily absent. Finally, the verifier should enforce *active_until* at authorization time, independently of the *is_active* flag.

IVP-09-004 WP4: Stale manifest replay accepted within validity window *(Low)*

Cure53 ascertained that the verifier's only freshness control rejects manifests with historical *valid_until* timestamps. The construct fails to maintain manifest sequence numbers or nonces, neglects to compare against the previously-applied manifest, and does not track the random manifest ID. Accordingly, a manifest will remain valid and replayable for the entire three-hour window between its *created_at* and *valid_until* timestamps.

An attacker able to serve manifests to the verifier (i.e., a compromised distributor, writeable manifest volume, or network position on a plaintext *MANIFEST_URL*) can replay an older yet valid manifest to roll back or freeze MailX subscription state. For example, the adversary can reinstate a tier that has been recently downgraded or ensure that a lapsed subscription remains active for up to three hours per replay. The lack of monotonicity enforcement means that the verifier cannot detect its backward movement.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected file:
services/verifier/service/service.go

Affected code:
```
if m.ValidUntil.Before(time.Now()) {
    log.Printf("manifest is expired: %v", m.ValidUntil)
    return fmt.Errorf("manifest is expired")
}
```

To mitigate this vulnerability, Cure53 advises incorporating a strictly increasing sequence number to the manifest that is covered by the signature. The verifier must persist the highest sequence applied and reject all older manifests. In addition, the validity window can be shortened to reduce the probability of replay attacks.

IVP-09-007 WP3: SQL injection via manifest-derived tier in MySQL path *(Medium)*

Cure53 noted that the MySQL subscription store assembles its UPDATE statement with *fmt.Sprintf* and a *strings.Builder*. The subscription ID, *is_active*, *active_until*, and tier are directly interpolated into the SQL text with *%s* quoting. However, this process lacks escaping or parameter binding. The tier value is manifest-derived: it originates from *account.Product* and passes unmodified through the generator's default branch for any product outside the two renamed cases. Therefore, a crafted tier breaks out of the tier CASE expression.

The testers reproduced the construction with the value *x' END, is_active = true WHERE '1'='1';* –. This endeavor confirmed that the *CASE* is closed, an attacker-controlled clause is appended that sets all rows to active, and the remainder of the statement is commented out. The local row ID is interpolated in the same unescaped manner.

Albeit, a crafted tier only reaches this sink inside a validly signed manifest. As such, successful exploitation either requires a forged manifest (IVP-09-001), whereby the attacker gains arbitrary SQL execution against the MailX subscription database, or a product name in the IVPN catalogue that contains SQL metacharacters. The PostgreSQL and MongoDB backends parameterize the same operation and remain unaffected by this situation, verifying the correctness of the template.

Affected file:
services/verifier/repository/subscription.go

Affected code:
```
tierCases.WriteString(fmt.Sprintf("WHEN '%s' THEN '%s' ", id, sub.Tier))
[...]
sql := fmt.Sprintf(`... tier = CASE id %s END WHERE id IN (%s);`, ...,
tierCases.String(), strings.Join(ids, ","))
return d.Client.Exec(sql).Error
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

To mitigate this issue, Cure53 advises replacing the string-built *CASE* expression with parameterized statements. This can be achieved by implementing a per-row *GORM* Updates inside a transaction (in correlation to the PostgreSQL path) or a bulk operation with bound parameters. If a single-statement *CASE* is required for performance, all interpolated values must be passed as bound query parameters, rather than formatted into the SQL text.

IVP-09-008 WP3: Unbounded gzip decompression enables DoS *(Medium)*

Cure53 ascertained that the verifier reads the entire response body and decompresses it with *gzip.NewReader* followed by *io.ReadAll* during manifest fetching. However, this process lacks both an *io.LimitReader* and a maximum manifest size. Neither side is bounded; the *fasthttp* client applies its response body size check solely when the configured limit is greater than zero, while *fiber.Get* utilizes the zero-limit default client, meaning that the compressed body is also uncapped. The auditors confirmed that a gzip payload of repetitive data comprising approximately 100KB expands to 100MB in memory, representing a 1028-fold amplification. Moreover, a nested or larger payload will reach an arbitrary size.

A malicious/compromised distributor or network attacker on a plaintext *MANIFEST_URL* (IVP-09-011) can return a small body that expands until the verifier exhausts memory and is terminated. The verifier is the only component that applies subscription updates; thus, its repeated failure freezes MailX subscription state, forcing a Denial-of-Service (DoS) condition.

Affected file:
services/verifier/client/http/http.go

Affected code:

```
reader, err := gzip.NewReader(bytes.NewReader(body))
[...]
decompressed, err := io.ReadAll(reader)
```

To mitigate this vulnerability, Cure53 advises wrapping the decompressor in an *io.LimitReader* with a finite ceiling that matches a realistic maximum manifest size. Additionally, *MaxResponseBodySize* should be configured on the HTTP client to bind the compressed body, while bodies that exceed either limit prior to allocation must be rejected.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers any and all noteworthy findings that did not incur an exploit but may assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy method by which to be called. Conclusively, while a vulnerability is present, an exploit may not always be possible.

IVP-09-003 WP4: SHA-512 pre-hash reversible by HSM provider *(Low)*

To ensure that IVPN account identifiers avoid the HSM provider, the token service hashes each account ID with SHA-512 prior to HSM submission. The service documentation states that the raw identifier is never sent outside IVPN, not even to the HSM. The pre-hash is unsalted and unkeyed. Furthermore, the IVPN account IDs follow the fixed public format *i-XXXX-XXXX-XXXX* over a 36-character alphabet, a space comprising approixmately 2^{62}. Resultingly, any party with access to the HSM request data can confirm a specific account with a single SHA-512 computation or recover the full set via enumeration. The HSM additionally returns the MAC, from which the manifest pseudonym *token_hash = base64(SHA-256(base64(mac)))* directly follows. As such, the provider links each account digest to its pseudonym, even without input inversion.

The HSM provider or any party with access to its request and response data (via telemetry, a breach, or legal compulsion) can reconstruct the mapping between IVPN account IDs and manifest pseudonyms, then confirm whether a targeted account is a subscriber.

This vector does not reveal the corresponding MailX user, as the link additionally requires read access to the MailX subscription store. However, the HSM provider is granted the same account-to-pseudonym mapping that the pre-hash is designed to withhold. Therefore, a Fortanix-plus-MailX path to full deanonymization is added alongside the intended IVPN-plus-MailX variant.

Affected file:
services/token/client/signer_fortanix.go

Affected code:

```
digest := sha512.Sum512([]byte(input))
data := sdkms.Blob(digest[:])
[...]
res, err := s.Client.Mac(ctx, req)
```

To mitigate this issue, Cure53 advises replacing the plain pre-hash with a keyed counterpart under a secret that never leaves IVPN. For example, *HMAC-SHA256(K_pre, accountID)* whereby *K_pre* is held by the token service exclusively, ensuring that the value submitted to the HSM cannot be linked back to an account identifier. Alternatively, token generation can be transferred to an HSM operated by IVPN.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

IVP-09-005 WP4: Single dual-purpose key lacks rotation & versioning *(Low)*

Cure53 confirmed that the system relies on a symmetric HMAC-SHA256 key that serves both per-account token generation and manifest signing. Neither the tokens nor manifests carry a key identifier or version field. The SHA-512 and HMAC-SHA256 algorithms are hardcoded on both the signing and verification implementations. Consequently, a rotation protocol has not been established, as replacing the key simultaneously invalidates every pseudonym stored by MailX and all in-flight manifests. Both an overlap period and per-token expiration are omitted.

A graceful recovery path is not provided if the key is suspected to be compromised, as rotation requires a synchronized flag-day across IVPN and MailX, as well as full re-enrollment of every subscriber (during which verification fails). The root cause underpinning the ability for token generation credentials to sign manifests can be attributed to the dual-purpose use of one key (IVP-09-001). The absence of algorithm agility means that any future migration away from SHA-256 or SHA-512 will be confronted with the same flag-day constraint.

Affected file:
services/token/client/signer_fortanix.go

Affected code:

```
alg := sdkms.DigestAlgorithmSha256
req := sdkms.MacRequest{
    Data: data,
    Alg:  &alg,
    Key:  sdkms.SobjectByID(keyId),
}
```

To mitigate this issue, Cure53 advises separating the token generation and manifest signing keys, integrating a key identifier field to both tokens and manifests, and ensuring that the verifier retains a small set of keys. This revised approach will ensure that new keys can be introduced with an overlap window, rather than a flag-day cutover. Lastly, routine rotation should be scheduled and a key compromise recovery procedure should be documented.

IVP-09-006 WP1: Absent rate limiting enables HSM DoS *(Low)*

The testing team determined that the pre-authorization service exposes */v1/preauth/add* and */v1/preauth/get/:id* without rate limiting on either endpoint. A rate limit helper is implemented in the Redis repository (Incr) and declared in the service Cache interface. However, it is never called anywhere in the codebase. Each */add* call performs an HSM MAC operation via the token service and one outbound webhook. Neither the preauth path nor token service throttle these calls. This activity contrasts with the manifest generator, which protects its own signing path with a *rate.Limiter*.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Accordingly, any holder of *PREAUTH_ADD_PSK* can drive the token service (and in turn the HSM) at an unbounded request rate, causing DoS or excessive HSM cost. The same absence of throttling allows the token signing oracle detailed in IVP-09-001 to exercise at line rate. The */get/:id* endpoint is similarly unthrottled, although its unguessable UUID identifier, in tandem with the PSK and IP checks, limit the value of brute force attacks in this context.

Affected file:
services/preauth/repository/redis.go

Affected code:

```
// implemented and part of the Cache interface, but never invoked
func (c *Redis) Incr(ctx context.Context, key string, expiration
time.Duration) error {
    err := c.Client.Incr(ctx, key).Err()
    if err != nil {
        return err
    }
    if expiration > 0 {
        return c.Client.Expire(ctx, key, expiration).Err()
    }
    return nil
}
```

To mitigate this issue, Cure53 advises integrating a rate limiter into both endpoints keyed per caller via the existing Incr hook or a request middleware. Moreover, a global throughput cap should be asserted on the token service to ensure that a single caller cannot saturate the HSM.

IVP-09-009 WP4: Cryptography disabled and non-safeguarded by default *(High)*

The Cure53 analysts verified that the repository's sample configuration, which the README instructs operators to copy into place, ships every cryptographic control disabled. Specifically, *TOKEN_MOCK=true* and *GENERATOR_MOCK=true* select keyless token generation and manifest signing; *TOKEN_TLS_ENABLED=false* removes mTLS from the token service; and *REDIS_TLS_ENABLED=false* and *CLIENT_PGSQL_SSLMODE=disable* select plaintext data stores.

In mock mode, the signer returns the keyless and freely computable *base64(SHA-512(input)),* while the verifier reads the same *TOKEN_MOCK* variable and verifies with a keyless string comparison. A runtime guard to tie mock mode or the plaintext transport to a non-production context is not enforced. Configuration validation only requires HSM credentials in the event that *TOKEN_MOCK=false*. Therefore, mock mode initiates silently; the only observable factor is a (mock) string in the verifier log.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

A deployment that follows the documented setup (i.e., copy the sample, set the pre-shared keys required for the services to start, and bring the stack up) but does not actively override the remaining defaults essentially runs without cryptographic security. Any individual able to deliver a manifest to the verifier can forge it without credentials. Moreover, the account-to-pseudonym mapping will be publicly computable, meaning that both unlinkability and unforgeability collapse.

While the finding is contingent on the operator refraining from overriding the defaults, the insecure state is the shipped default and the omission may not be caught.

Affected file:
.env.sample

Affected code:
```
TOKEN_MOCK=true
TOKEN_TLS_ENABLED=false
GENERATOR_MOCK=true
```

To mitigate this issue, Cure53 advises ensuring that the insecure states are non-triggerable in production. The services should refuse to initiate in mock mode unless an explicit development flag is set, which will force the default to fail closed rather than open. The sample should ship secure values and transfer the development values to a clearly-marked development file. Moreover, the token service should require mTLS by default with the insecure fallback removed from the startup path. Lastly, production monitoring should provide an alert for any (mock) verification.

IVP-09-010 WP3: HSM gRPC lacks mandatory mTLS authorization *(Medium)*

Cure53 verified that the token service is the only component capable of invoking the HSM MAC key. Its gRPC endpoint is intended to be protected by mutual TLS against a private certificate authority. However, two weaknesses were observed that undermine this assurance.

Firstly, in the event that *TOKEN_TLS_ENABLED=false* is configured, the service initiates in plaintext with a log warning solely. The associated clients refuse to connect unless the *GENERATOR_ALLOW_INSECURE* or *PREAUTH_ALLOW_INSECURE* escape hatch is set. These are read by raw os.Getenv outside configuration validation. The certificate generation script referenced by the sample is absent from the repository; as a result, enabling the escape hatch represents the path of least resistance.

Secondly, the service fails to perform per-certificate authorization, even with mTLS enabled. Any certificate that chains to the private CA may call Generate with arbitrary input. The manifest signing generator and Internet-exposed preauth service (which should only mint account tokens) will be indistinguishable to the server.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

In the escape hatch configuration, the HSM chokepoint constitutes an unauthenticated oracle to any peer on the internal network. Due to the omission of per-certificate authorization, the preauth service's certificate implicitly represents a manifest signing credential. Thus, a compromise of the exposed preauth service yields the full signing oracle (discussed in ticket IVP-09-001).

Affected file:
services/token/service/service.go

Affected code:
```
} else {
    log.Println("WARNING: Token service is running without TLS — do not use
in production")
}
```

To mitigate this issue, Cure53 advises requiring mTLS in production without a plaintext fallback. Moreover, the transport decision should be transferred into configuration validation to ensure that it fails closed. Furthermore, a functional certificate provisioning path should be shipped. Lastly, each client certificate should be authorized to a distinct set of operations to guarantee that token minting credentials cannot sign manifests.

IVP-09-011 WP3: Manifest fetch lacks TLS or pin distributor enforcement *(Medium)*

The consulting team observed that the verifier fetches the manifest from *MANIFEST_URL* with a bearer *MANIFEST_PSK*. However, the URL configuration neglects to enforce *https* or certificate pinning. If configured as *http*, the pre-shared key is transmitted in cleartext upon each hourly fetch and the manifest is retrieved over an unauthenticated channel.

The manifest signature protects integrity, blocking Man-in-the-Middle (MitM) attackers from forging content in the non-mock configuration. Nevertheless, plausible ramifications of this situation include harvesting *MANIFEST_PSK*, replaying of older yet valid manifests (IVP-09-004), and delivery of decompression bombs (IVP-09-008).

This pitfall degrades *MANIFEST_PSK* confidentiality and grants MitM adversaries replay and DoS capabilities. The exposure is deployment dependent and is limited to the absence of pinning if *MANIFEST_URL* already constitutes an HTTPS endpoint with a trusted certificate authority.

Affected file:
services/verifier/client/http/http.go

Affected code:
```
req := fiber.Get(h.Cfg.ManifestURL)
req.Set("Authorization", "Bearer "+h.Cfg.ManifestPSK)
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

To mitigate this issue, Cure53 advises rejecting any non-*https MANIFEST_URL* upon configuration validation. The distributor's certificate should be pinned or restricted to a known CA.

IVP-09-012 WP3: Lenient token parsing *(Low)*

During the code review, Cure53 determined that token verification depends on properties of the JSON parser. In general, token parsing is conducted using Golang's standard JSON library. However, the parser only performs a minimal number of checks and fails to detect fields that are present multiple times. Moreover, unknown fields are silently dropped. The default JSON parser in Golang is not case sensitive and only considers field names using different cases as matches. The majority of JSON parsers are case sensitive.

The code parses a token, then removes the signature and re-serializes the remaining fields. The serialized result is subsequently leveraged as data for signature verification. Parsing and serializing via a lenient parser enables modification of the original token, e.g., by adding unknown fields or duplicating existing variants. Furthermore, parsing and serializing removes the modifications; therefore, modified tokens still pass signature verification.

Affected file #1:
services/distributor/service/service.go

Affected code #1:
```
// Unmarshal JSON into Manifest struct
var manifest model.Manifest
if err = json.Unmarshal(bytes, &manifest); err != nil {
   return model.Manifest{},
       fmt.Errorf("failed to unmarshal manifest: %w", err)
}
```

Affected file #2:
services/verifier/client/http/http.go

Affected code #2:
```
// Unmarshal JSON into Manifest struct
var manifest model.Manifest
if err = json.Unmarshal(bytes, &manifest); err != nil {
   return model.Manifest{},
       fmt.Errorf("failed to unmarshal manifest: %w", err)
}
```

Token modification is possible via field addition or duplication, forcing alternate parsers to interpret the same token via different methods. This circumstance is exemplified by CVE 2026-27896, in which the primary concerns entail "*bypassing intermediary inspection*" and "*cross-implementation inconsistency*".

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Albeit, Cure53's evaluation determined that critical fields (i.e., those included in the manifest) cannot be altered. Accordingly, a viable exploit pathway could not be located.

To mitigate this issue, Cure53 suggests applying stricter token parsing. An effective solution would be to utilize *json/v2*[1], since this version supports stricter parsing via granular options such as *RejectUnknownMembers*, which ensures that the JSON string does not contain additional fields. The old version of the JSON parser allows duplicate fields, which does not conform to recommended guidance.[2] *json/v2* fixes this behavior and only permits duplicate fields using the *AllowDuplicateNames* option**.**

MatchCaseInsensitiveNames can be adopted to define name case sensitivity; JSON is typically case sensitive, yet the Golang default is converse. Field case sensitivity will help to ensure that tokens remain unambiguous.

An alternative countermeasure would be to marshal the parsed structure following unmarshalling and verify whether the original JSON string has been obtained. For this approach, the structure's JSON serialization must be deterministic and stable over multiple versions. Deterministic JSON serialization should not be considered an additional requirement, since signature verification is based on the assumption that serialization is deterministic and stable.

[1] https://pkg.go.dev/encoding/json/v2
[2] https://www.rfc-editor.org/rfc/rfc7493.html#section-2.3

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Conclusions

This section provides a detailed discussion of the overall verdict gathered by the Cure53 team during this summer 2026 evaluation of the IVPN Unlinked Service Access feature. To summarize, the inspected components garnered a relatively positive impression, demonstrating a decent structural baseline that nonetheless exhibits leeway for improvement.

In terms of the coverage and findings, the project was vetted for key reuse and ambiguous inputs for signatures. Crucially, the signed data must contain sufficient context to avoid misinterpretation elsewhere. To avoid these situations, cryptographic keys should only be used for a single purpose. However, an HMAC key is utilized in several situations, introducing the viability of signature forgery (IVP-09-001). Cure53 recommends either deriving distinct keys for unique purposes or extending signed data with additional fields that unambiguously describe the full context of the signature.

Onward, the plausibility of information disclosure was estimated. Here, it was noticed that hash functions are used to hide information, yet IVP-09-003 confirms that hashing input (rather than leveraging a keyed hash) can lead to disclosure.

Generally speaking, replay attacks and message reordering can lead to message intent amendments. In this realm, Cure53 determined that the premise lacks sufficient protection against these risks (IVP-09-004). While the inclusion of timestamps and expiration dates is advantageous, extra countermeasures should be employed to ensure a consistent database state.

During the evaluation of input verifications, Cure53 confirmed that the code is susceptible to SQL injection attacks (IVP-09-007) due to the omission of an input format check. Since this vector is a recurring problem, most SQL clients feature APIs that facilitate proper formatting. APIs of this nature should be adopted to negate correlating risks, rather than persisting with query formatting in the code.

Cure53 assessed the project's general structure and resistance to malicious usage. Upon review, the confirmation was made that one could switch into test/mock mode and disregard the settings (IVP-09-009). Weak settings are established by default and can be overlooked, owing to the lack of appropriate safeguards. Best practice mandates configuring defaults to the strongest available option and only allowing security check circumvention upon explicit request.

The likelihood of DoS attacks was considered, unearthing two connected defects requiring amelioration. IVP-09-006 details a lack of rate limiting parameters, which should be integrated to prevent DoS attacks, as well as assist with detecting misconfigurations and misbehaving code. IVP-09-008 verifies the absence of gzip decompression memory limits, thus facilitating system inoperation.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Cryptographic agility was also covered. Projects should generally be designed for straightforward key rotation, modification of existing cryptographic schemes, and the addition of new primitives. Failure to impose cryptographic agility means that minor weaknesses detected during a project's ongoing production may remain unfixed, since alterations may evoke undesirable disruption. IVP-09-005 confirms the lack of an easy key rotation procedure and protocol version; these should be installed in the early stage of development to simplify remediation efforts during production.

Logical flaws and fundamental step omissions were vetted. Here, an issue was located whereby subscription revocation is inadequately propagated (IVP-09-002). Elsewhere, Cure53 detected two TLS deficiencies, including a lack of both mTLS authorization and a TLS/pin distributor (IVP-09-010 and IVP-09-011, respectively). The key risk here is that the solution can be configured securely in theory, yet the setup is not enforced by the code, fostering brittle implementations. Stricter configuration verification should be assured to avoid insecure deployments.

During the code review, Cure53 appraised Golang's JSON parser. The implementation is effective for formatting digital signature and MAC inputs, ensuring that the inputs remain unambiguous.

Cure53 observed that verification first parses the JSON-extracted information, then serializes and compares it against the provided HMAC. This process differs from the verification approach leveraged in JWT, which utilizes and validates an unmodified string adopted from the token. The process of deserialization and serialization imbues signature malleability to the token verification process. Specifically, Golang's JSON parser is lenient and allows a number of token modifications of the tokens that are neither detected nor rejected (IVP-09-012). The latest version (v2) should be applied in tandem with strict options that limit serialized token alterations as much as feasible.

Token modification inherently expands the attack surface. JSON features non-deterministic elements and the Golang parser deviates from the standard on occasion; for instance, via the neglect to enforce case sensitivity. As such, an attacker could generate tokens that are interpreted by JSON parsers in alternate ways, depending on which token elements are matched with a given field in the target structure and those that are discarded. This attack vector was cross-referenced with similar pathways. Primary concerns included token inspection bypassing and unique token interpretation by alternate implementations or versions. In summary, Cure53 concluded that the data received via token verification (i.e., manifest) matches the originally signed counterpart. As a result, no possible exploit avenues could be identified in this area.

The underlying cryptographic primitives and parameters were examined, confirming state-of-the-art paradigms that minimize the opportunity for incorrect use (aside from potential invalid formatting). Cure53 sought to detect any weak or outdated cryptographic primitives and libraries, ultimately to no avail.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Upon concluding the designated objectives for this July 2026 engagement, Cure53 has formed a favorable assessment of the IVPN Unlinked Service Access feature, qualified only by a limited number of vulnerabilities that require resolution prior to production deployment. The four scoped work packages received comprehensive white-box analysis at both the source-code and runtime-system levels. The twelve documented findings are distributed across the token, preauth, distributor, and verifier services, rather than concentrated within any single component. Of those, five are classified as security vulnerabilities and seven as general weaknesses, with only two assigned a *High* severity rating. Notably, neither of these findings stem from compromised cryptographic primitives or fundamental protocol design flaws. Rather, both originate from integration and deployment anomalies regarding the method by which otherwise secure building blocks are combined and deployed.

The cryptographic core of the system is appropriate and cutting-edge, yet its guarantees are eroded by insufficient domain separation and a configuration surface that permits (and sometimes defaults to) an insecure posture. The single most impactful finding verifies that an HMAC key serves two distinct purposes without context binding, which dissolves the boundary between minting an account token and signing a manifest (IVP-09-001). This converts a token generation credential into a manifest forgery oracle. Similarly, IVP-09-009 outlines that the entire stack and cryptographic controls are disabled by default. Moreover, transport protection can be quietly downgraded; a system fully capable of secure operation is not compelled by its own code to remain in this state (IVP-09-010 and IVP-09-011). Every signature should be bound to an explicit and unambiguous context, with keys ideally separated by purpose (IVP-09-005). The secure configuration should become the enforced default, thus ensuring that the software fails closed rather than open.

The codebase is compact, readable, and internally consistent. The selected primitives grant scant opportunity for misuse, while the privacy and unlinkability model that motivates the feature is thoughtfully constructed. The residual exposure is confined to manageable edge-case scenarios, such as the reversible HSM pre-hash described in IVP-09-003. The remaining *Medium*- and *Low*-severity discoveries encompass input validation and parameterized queries, replay and reordering protection, decompression and rate limiting, key rotation and versioning, and stricter token parsing. The corresponding tickets should be individually addressed, yet do not reflect systemic weaknesses. The actionable fix advice provided for each pitfall should be straightforward to implement.

Following the resolution of the major shortcomings, as well as the integration of key separation and fail-closed configurations, the Unlinked Service Access feature will verifiably offer a powerful security foundation.

Cure53 would like to thank Juraj Hilje and Viktor Vecsei from the IVPN Limited team for their excellent project coordination, support, and assistance, both before and during this assignment.