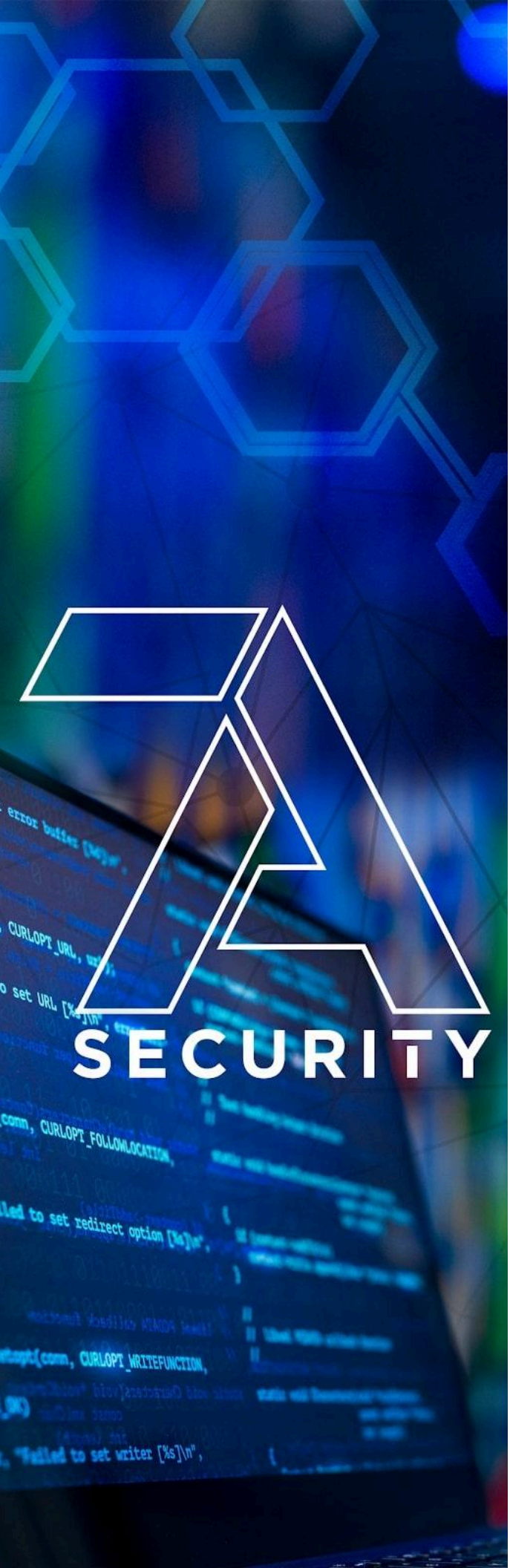


PLATINUM

ISO/IEC 27001:2022
ISMS Certified
by Consilium Labs (IAS)

AICPA
SOC 2
Formerly SAS 70 Reports

Pentest Report

Client: *Stork Team*

in collaboration with the

Open Source Technology Improvement Fund, Inc.

Stork Test Targets:

Web Frontend & API
Backend, Agents & Auth
Stork Supply Chain
Threat Model
Dependency & SBOM
Config Parser Fuzzing

7ASecurity Test Team:

- Abraham Aranguren, MSc.
- Daniel Ortiz, MSc.
- Dariusz Jastrzębski
- Dheeraj Joshi, BTech.
- Miroslav Štampar, PhD.
- Szymon Grzybowski, MSc.
- Patrick Ventuzelo, MSc.
- Nabih Benazzouz, MSc.

7ASecurity
Protect Your Site & Apps
From Attackers
sales@7asecurity.com
7asecurity.com

INDEX

Introduction

“Stork is an open source management tool for ISC's open source software systems.”
From https://stork.isc.org/

This document outlines the results of a penetration test and *whitebox* security review conducted against the Stork platform. The project was solicited by the Stork maintainers, facilitated by the *Open Source Technology Improvement Fund, Inc (OSTIF)*, funded by the *Internet Systems Consortium (ISC)*, and executed by 7ASecurity in September and October 2025. The audit team dedicated 36.57 working days to complete this assignment. Please note that this is the second penetration test for this project, following an internal audit by an ISC engineer unrelated to the Stork development team, around 2023. Additionally, Stork has been receiving, reviewing and fixing findings from security researchers for a number of years. Consequently, the identification of security weaknesses was expected to be more difficult during this engagement, as more vulnerabilities are identified and resolved after each testing cycle.

During this iteration the goal was to review the solution as thoroughly as possible, to ensure Stork users can be provided with the best possible security. The methodology implemented was *whitebox*: 7ASecurity was provided with access to a staging environment, documentation, test users, and source code. A team of 8 senior auditors carried out all tasks required for this engagement, including preparation, delivery, documentation of findings and communication.

A number of necessary arrangements were in place by September 2025, to facilitate a straightforward commencement for 7ASecurity. In order to enable effective collaboration, information to coordinate the test was relayed through email, as well as a shared Mattermost Channel. The Stork team was helpful and responsive throughout the audit, which ensured that 7ASecurity was provided with the necessary access and information at all times, thus avoiding unnecessary delays. 7ASecurity provided regular updates regarding the audit status and its interim findings during the engagement.

The audit was split across the following work packages:

- WP1: Whitebox Tests against Stork Web Frontend and API
- WP2: Whitebox Tests against Stork Backend, Agents & Auth Logic
- WP3: Whitebox Tests against Stork Supply Chain Implementation
- WP4: Stork Lightweight Threat Model Documentation
- WP5: Stork Dependency Management Review & SBOM Creation
- WP6: Stork Config Parser Fuzzing and Test Case Creation

The findings of the security audit (WP1-2, WP6) can be summarized as follows:

Identified Vulnerabilities	*Hardening Recommendations*	*Total Issues*
7	*5*	*12*

Please note that the results of WP3 and WP4 are described in the following report sections:

- WP3: Stork Supply Chain Implementation
- WP4: Stork Lightweight Threat Model

Regarding *WP5: Stork Dependency Management Review & SBOM Creation*, a separate analysis and SBOM was provided to the Stork team.

Moving forward, the scope section elaborates on the items under review, while the findings section documents the identified vulnerabilities followed by hardening recommendations with lower exploitation potential. Each finding includes a technical description, a proof-of-concept (PoC) and/or steps to reproduce if required, plus mitigation or fix advice for follow-up actions by the development team.

Finally, the report culminates with a conclusion providing detailed commentary, analysis, and guidance relating to the context, preparation, and general impressions gained throughout this test, as well as a summary of the perceived security posture of the Stork applications.

About OSTIF

The *Open Source Technology Improvement Fund (OSTIF)* is dedicated to resourcing and managing security engagements for open source software projects through partnerships with corporate, government, and non-profit donors. We bridge the gap between resources and security outcomes, while supporting and championing the open source community whose efforts underpin our digital landscape.

Over the past ten years, OSTIF has been responsible for the discovery of over 800 vulnerabilities, (121 of those being Critical/High), over 13,000 hours of security work, and millions of dollars raised for open source security. Maximizing output and security outcomes while minimizing labor and cost for projects and funders has resulted in partnerships with multi-billion dollar companies, top open source foundations, government organizations, and respected individuals in the space. Most importantly, we've helped over 120 projects and counting improve their security posture.

Our directive is to support and enrich the open source community through providing public-facing security audits, educational resources, meetups, tooling, and advice. OSTIF's experience positions us to be able to share knowledge of auditing with maintainers, developers, foundations, and the community to further secure our infrastructure in a sustainable manner.

We are a small team working out of Chicago, Illinois. Our website is ostif.org. You can follow us on social media to keep up to date on audits, conferences, meetups, and opportunities with OSTIF, or feel free to reach out directly at contactus@ostif.org or our Github.

Derek Zimmer, Executive Director
Amir Montazery, Managing Director
Helen Woeste, Communications and Community Manager
Tom Welter, Project Manager

Scope

The following list outlines the items in scope for this project:

- **WP1: Whitebox Tests against Stork Web Frontend and API**
  - **Main Focus:** Refactoring Branch (new upcoming code) https://gitlab.isc.org/isc-projects/stork/-/tree/1760-upgrade-primeng-2
  - **Secondary Focus:** For reference purposes only https://gitlab.isc.org/isc-projects/stork/-/releases/v2.3.0
  - **Documentation:** https://stork.isc.org/
- **WP2: Whitebox Tests against Stork Backend, Agents & Auth Logic**
  - **Main Focus:** Refactoring Branch (new upcoming code) https://gitlab.isc.org/isc-projects/stork/-/tree/1835-kea-3-0-...ent
  - **Secondary Focus:** For reference purposes only https://gitlab.isc.org/isc-projects/stork/-/releases/v2.3.0
- **WP3: Whitebox Tests against Stork Supply Chain Implementation**
  - Information about release procedures was provided by Stork
- **WP4: Stork Lightweight Threat Model Documentation**
  - Information was provided by the Stork team
- **WP5: Stork Dependency Management Review & SBOM Creation**
  - https://gitlab.isc.org/isc-projects/stork/
- **WP6: Stork Config Parser Fuzzing and Test Case Creation**
  - https://gitlab.isc.org/isc-projects/stork/-/tree/master/backend/agent
    - **bind9:** BIND9 configuration parser
    - **pdns:** PowerDNS configuration parser
    - **kea:** JSON-based Kea configuration parser with custom unmarshalling
  - https://gitlab.isc.org/isc-projects/stork/-/tree/master/backend/appcfg
    - **monitor.go:** Functions parsing configuration fragments using regular expressions

Identified Vulnerabilities

This area of the report enumerates findings that were deemed to exhibit greater risk potential. Please note these are offered sequentially as they were uncovered, they are not sorted by significance or impact. Each finding has a unique ID (i.e. *STO-01-001*) for ease of reference, and offers an estimated severity in brackets alongside the title.

STO-01-002 WP2: Remote Denial of Monitoring via Kea Statistics *(Medium)*

Retest Notes: Resolved by Stork, and verified by 7ASecurity.

The Stork agent is vulnerable to a remote denial of monitoring condition due to improper validation when parsing statistics from a Kea DHCP server. An attacker with administrative control over a monitored Kea instance can prevent the agent from reporting Kea-related metrics by sending a single, maliciously crafted statistic name. The agent JSON parser assumes that a successful string prefix match guarantees a valid regular expression match. This leads to a *nil* pointer dereference, which crashes the Prometheus metrics handler and persistently disables monitoring for that Kea instance.

This vulnerability significantly affects the integrity and reliability of the monitoring system. An attacker can exploit this flaw to obscure malicious activity, such as distributing rogue DNS servers through DHCP. Although the agent process remains active, every automated poll for Kea statistics fails, creating a persistent silent failure. The monitoring dashboard becomes non-functional, delaying detection and incident response.

The issue originates from missing validation in the custom JSON unmarshaling logic. The parser for Kea statistics correctly identifies subnet-related entries by checking for the *“subnet[”* prefix but immediately applies a regular expression without verifying a successful match. A statistic name such as *“subnet[invalid].metric”* passes the prefix check but fails the regex, causing *regexp.FindStringSubmatch* to return *nil*. Accessing an index on this *nil* slice triggers a runtime panic that terminates the handler and prevents statistics delivery.

Exploitation requires administrative control of a Kea server and injection of a malformed statistic. This may be obtained via the Kea hook system by installing a Python or C++ hook[1] that modifies responses to the *statistic-get-all command*. The hook would inject a malformed key-value pair such as *"subnet[invalid].metric": [[0, "2025-09-28T12:00:00Z"]]* into the JSON body before transmission to the Stork agent. During subsequent Prometheus scrapes, the agent receives this data, triggers a panic, and fails to collect any Kea metrics.

[1] https://kea.readthedocs.io/en/latest/arm/hooks.html

Affected File:
https://gitlab.isc.org/isc-projects/stork/[...]/backend/agent/promkeaexporter.go

Affected Code:
```
// Collect stats from all Kea apps.
func (pke *PromKeaExporter) collectStats() error {
    [...]
    for _, app := range apps {
        [...]
        // Fetching statistics
        responseData, err := keaApp.sendCommandRaw(requestDataBytes)
        if err != nil {
            lastErr = err
            log.WithError(err).Error("Problem fetching stats from Kea")
            continue
        }

        // Parse response
        var response keactrl.StatisticGetAllResponse
        err = json.Unmarshal(responseData, &response)
        if err != nil {
            lastErr = err
            log.WithError(err).
                WithField("request", requestData).
                Error("Failed to parse responses from Kea")
            continue
        }
```

Affected File:
https://gitlab.isc.org/isc-projects/stork/[...]/backend/appctrl/kea/statcmds.go

Affected Code:
```
const (
    StatisticGet    CommandName = "statistic-get"
    StatisticGetAll CommandName = "statistic-get-all"
)

var (
    // Matches a subnet ID in the Kea statistic name.
    subnetStatNameRegex = regexp.MustCompile(`subnet\[(\d+)\]\.(.+)`)
    // Matches a pool ID in the Kea statistic name.
    poolStatNameRegex = regexp.MustCompile(`pool\[(\d+)\]\.(.+)`)
    // Matches a prefix pool ID in the Kea statistic name.
    pdPoolStatNameRegex = regexp.MustCompile(`pd-pool\[(\d+)\]\.(.+)`)
)

[...]
```

```
// UnmarshalJSON implements json.Unmarshaler. It unpacks the Kea response
// to simpler Go-friendly form.
func (r *StatisticGetAllResponseArguments) UnmarshalJSON(b []byte) error {
    [...]
    var obj map[string][][]json.RawMessage

    err := json.Unmarshal(b, &obj)
    if err != nil {
        err = errors.Wrapf(err, "failed to parse response arguments from Kea")
        return err
    }

    var samples []*StatisticGetAllResponseSample

    // Retrieve values of mixed-type arrays.
    // Unpack the complex structure to simpler form.
    for statName, statValueOuterList := range obj {
        var subnetID int64
        var addressPoolID *int64
        var prefixPoolID *int64
        // Extract the subnet ID and pool ID if present.
        if strings.HasPrefix(statName, "subnet[") {
            matches := subnetStatNameRegex.FindStringSubmatch(statName)
            subnetIDRaw := matches[1]
            statName = matches[2]
```

It is recommended to add validation immediately after each call to *FindStringSubmatch* to ensure the returned slice is not *nil* and has the expected length before accessing its elements. Malformed statistic names should be logged and skipped, allowing the agent to continue processing valid entries. This change will ensure resilience against both accidental and malicious malformed data from monitored services.

STO-01-003 WP2: Latent SQL Injection in stork-tool *(Low)*

Retest Notes: Resolved by Stork, and verified by 7ASecurity.

The *stork-tool* command-line utility is vulnerable to a latent SQL injection. This occurs because the *db-create* command constructs an SQL statement by directly embedding a raw, unsanitized database name provided by the user. Although a direct exploit is currently prevented by an incidental interaction between the database driver and a PostgreSQL rule, the vulnerability represents a dormant attack vector that may become exploitable if application dependencies or database behavior change, potentially allowing arbitrary command execution on the database server.

This arises from the use of *fmt.Sprintf* to construct the *CREATE DATABASE* SQL statement, which is an unsafe pattern. The injection attempt is currently mitigated by a

two-stage interaction: first, the database driver transmits the entire injected payload as a single command; second, the PostgreSQL server wraps the command in a transaction, which fails because *CREATE DATABASE* cannot be executed within a transaction block. Relying on this incidental protection is insecure, as any future modification, such as adding a DDL command without this restriction, or changing driver behavior, could make the code exploitable.

Additionally, the *stork-tool* exposes this flawed code path by automatically using credentials from environment variables. This allows a user with limited configuration access, such as the ability to modify CI/CD variable files, to trigger the unsafe logic using privileged credentials they cannot view directly.

Affected File:
https://gitlab.isc.org/isc-projects/stork/[...]/backend/cmd/stork-tool/main.go

Affected Code:

```
// Execute db-create command. It prepares new database for the Stork
// server. It also creates a user that can access this database using
// a generated or user-specified password and the pgcrypto extension.
func runDBCreate(context *cli.Context) {
    flags := &dbops.DatabaseCLIFlagsWithMaintenance{}
    flags.ReadFromCLI(context)
    var err error
    [...]
    // Try to create the database and the user with access using
    // specified password.
    err = dbops.CreateDatabase(
        *settings,
        flags.DBName,
        flags.User,
        flags.Password,
        context.Bool("force"),
    )
```

Affected File:
https://gitlab.isc.org/isc-projects/stork/[...]/database/maintenance/database.go

Affected Code:

```
// Create database with a given name.
func CreateDatabase(db *pg.DB, dbName string) (created bool, err error) {
    _, err = db.Exec(fmt.Sprintf("CREATE DATABASE %s;", dbName))
    if err != nil {
        var pgErr pg.Error
        if errors.As(err, &pgErr) && pgErr.Field('C') == "42P04" { //
duplicate_database
            return false, nil
        }
```

```
        err = errors.Wrapf(err, `problem creating the database "%s"`, dbName)
        return false, err
    }
    return true, nil
}
```

It is recommended to strictly validate all user-supplied identifiers before their inclusion in SQL statements. Since object names must appear directly within SQL syntax and cannot be parameterized, variables such as *dbName* should be validated against a restrictive allow-list regular expression, for example *^[a-zA-Z0-9_]+$.* This validation should be implemented in the low-level maintenance functions to ensure secure-by-design behavior and to prevent misuse by higher-level functions that operate with inherited privileged credentials. This will prevent injection of special characters or malicious SQL commands into the statement string.

STO-01-004 WP1: HTML Injection via Daemon Version *(Medium)*

Retest Notes: Resolved by Stork, and verified by 7ASecurity.

The Stork web interface is affected by a stored HTML injection vulnerability. The agent component accepts the extended version string reported by a monitored daemon, such as Kea DHCP, without validation. The server stores this value unsanitized, and the frontend renders it as raw HTML. This enables an attacker controlling a monitored daemon to inject arbitrary HTML into the administrative interface. Although the built-in Angular sanitizer blocks script execution, the flaw allows phishing and UI redressing attacks against administrators, potentially leading to credential theft and full account compromise.

This enables escalation from a single compromised daemon to full control of the central Stork management system. A malicious version string can inject deceptive HTML elements that imitate legitimate interface components, such as buttons or alerts. When an administrator views the compromised daemon, these elements appear authentic, for example a fake *“Session Expired”* button, which can redirect the victim to a phishing page and capture credentials.

The issue originates in the frontend, where the *[innerHTML]* binding is used to render the *extendedVersion* property of a daemon. The Stork agent retrieves this string through a version-get command sent to the monitored daemon, transmits it to the server, and stores it in the extended_version database column. When an administrator views the daemon details, the frontend retrieves the string, replaces newline characters with <*br*> tags, and passes it directly to *[innerHTML]*. This triggers the built-in Angular sanitizer but still allows rendering of safe tags such as <*a*> and <*b*>.

Conceptual PoC:
An attacker controlling a monitored Kea DHCP server modifies the daemon binary to report a malicious version string such as:

```
<a href="//attacker.com/stork-auth-reset" class="p-button
p-button-danger"><b>Session Expired - Re-Authenticate</b></a>
```

When the Stork administrator views the affected daemon, this payload appears as a red, high-priority button consistent with the Stork interface design. Clicking it redirects the administrator to an attacker-controlled phishing page, leading to credential compromise.

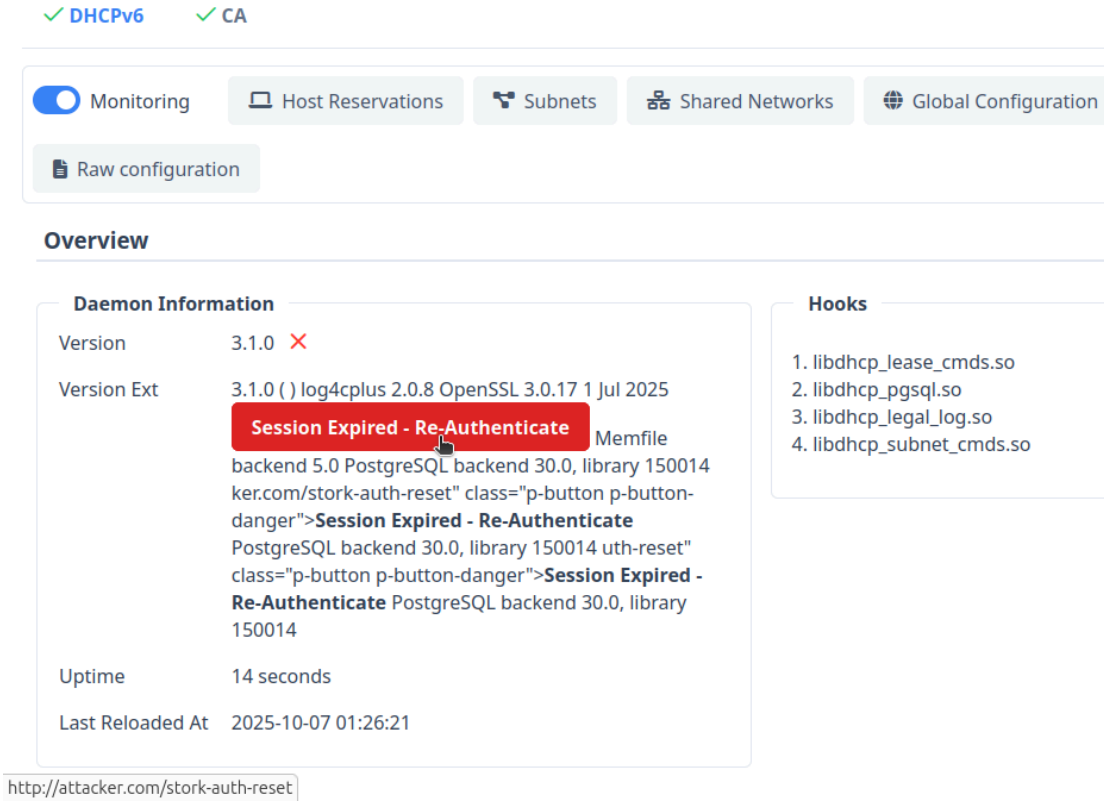


Fig.: Stork administrative interface manipulated via HTML injection

Affected File:
https://gitlab.isc.org/isc-projects/stork/[...]/app/kea-app-tab/kea-app-tab.component.html

Affected Code:

```
[...]
<div class="col-12 sm:col-3 pb-0 font-medium">Version Ext</div>
<div class="col-12 sm:col-9" [innerHTML]="daemon.extendedVersion"></div>
<div class="col-12 sm:col-3 pb-0 font-medium">Uptime</div>
[...]
```

Affected File:
https://gitlab.isc.org/isc-projects/stork/[...]/app/apps-page/apps-page.component.ts

Affected Code:

```
function htmlizeExtVersion(app: App) {
    if (app.details.extendedVersion) {
        app.details.extendedVersion = app.details.extendedVersion.replace(/\n/g,
'<br>')
    }
    if (app.details.daemons) {
        for (const d of app.details.daemons) {
            if (d.extendedVersion) {
                d.extendedVersion = d.extendedVersion.replace(/\n/g, '<br>')
            }
        }
    }
}
```

It is recommended to treat all data received from external services as untrusted plain text. The *[innerHTML]* binding should be replaced with standard string interpolation *({{ daemon.extendedVersion }})* to ensure that HTML characters are rendered as literal text rather than interpreted by the browser. This change eliminates the HTML injection vector entirely. As a defense-in-depth measure, a strict *Content Security Policy (CSP)* should also be implemented to further restrict the types of content that can be rendered within the application.

STO-01-009 WP6: DoS via Memory Allocation in PowerDNS Parser *(Medium)*

Retest Notes: Resolved by Stork, and verified by 7ASecurity.

The Stork server is vulnerable to a Denial-of-Service (DoS) condition caused by uncontrolled memory allocation during PowerDNS configuration parsing. An attacker who provides a malicious configuration file can trigger excessive memory allocation, resulting in an Out-of-Memory (OOM) panic that terminates the Stork server and impacts system availability.

The issue is located in the *parseLine* function of the PowerDNS configuration parser. The code preallocates a slice for parsed values using *make([]ParsedValue, 0, len(fields))*, where *len(fields)* represents the number of space-separated tokens in a single configuration line. Because no validation is performed on the number of fields before allocation, an attacker can craft a configuration line containing an extremely large number of tokens. This causes the parser to request memory in gigabyte or terabyte ranges, immediately leading to an OOM panic and service crash.

Affected File:
https://gitlab.isc.org/isc-projects/stork/-/blob/[...]/backend/appcfg/pdns/parser.go#L56

Affected Code:

```
// Parses the PowerDNS configuration from a reader.
func (p *Parser) Parse(reader io.Reader) (*Config, error) {
        // Define a map for storing key/values pairs. Note that there may be
        // multiple values separated by commas or spaces.
        parsedMap := make(map[string][]ParsedValue)

        // The default scanner splits the input into lines.
        scanner := bufio.NewScanner(reader)
        for scanner.Scan() {
                ...
                parsedValues := make([]ParsedValue, 0, len(fields))
```

PoC:
The following Go test demonstrates the underlying issue. The *make* function is invoked with a length derived directly from unvalidated user input, triggering an OOM error:

```
func TestMakeCrash(t *testing.T) {
    field := 1000000000000000
    parsedValue := make([]ParsedValue, 0, field)
    require.NotNil(t, parsedValue)
}
```

It is recommended to implement an upper limit on the number of fields processed per configuration line. Before allocating memory, the parser should verify that *len(fields)* does not exceed this threshold. Lines exceeding the limit should be rejected with an error rather than parsed. This validation prevents OOM conditions and ensures that the parser remains resilient against malformed or malicious inputs.

STO-01-010 WP6: DoS via Line Length in PowerDNS Parser *(Medium)*

Retest Notes: Resolved by Stork, and verified by 7ASecurity.

The Stork server is vulnerable to a Denial-of-Service (DoS) condition when parsing PowerDNS configuration files. The parser uses *bufio.Scanner* with its default settings, which cannot handle lines longer than 64 KB. A specially crafted configuration file containing an excessively long line can cause the scanner to fail or trigger excessive memory allocation, potentially leading to an Out-of-Memory (OOM) condition and server crash.

The issue is located in the *Parse* function of the PowerDNS configuration parser. The function reads configuration data using *bufio.NewScanner(reader)*, which defaults to a maximum token size of *bufio.MaxScanTokenSize*. When *scanner.Scan()* encounters a

line exceeding this limit, it returns *ErrTooLong*. While the error itself is handled, a single-line file with no newline characters can still cause large memory allocations before the limit is reached, resulting in resource exhaustion and service unavailability.

Affected File:
https://gitlab.isc.org/isc-projects/stork/-/blob/[...]/backend/appcfg/pdns/parser.go#L31

Affected Code:

```
// Parses the PowerDNS configuration from a reader.
func (p *Parser) Parse(reader io.Reader) (*Config, error) {
	// Define a map for storing key/values pairs. Note that there may be
	// multiple values separated by commas or spaces.
	parsedMap := make(map[string][]ParsedValue)

	// The default scanner splits the input into lines.
	scanner := bufio.NewScanner(reader)
	for scanner.Scan() {
[...]
```

PoC:
A configuration file (*pdns.conf*) containing a single line longer than 64 KB causes the parser to fail. A fuzzing script (*backend/appcfg/pdns/generate_large_pdns_config.py*) can be used to generate such files. When the parser processes this input, *scanner.Scan()* fails due to exceeding the token limit or attempts to allocate excessive memory, leading to a crash.

It is recommended to configure the scanner with a larger but bounded buffer size using *scanner.Buffer()* to support long but valid configuration lines. The code should also explicitly detect and handle *bufio.ErrTooLong* gracefully, treating it as a syntax error instead of allowing it to propagate and terminate the service.

STO-01-011 WP6: DoS via Nil Pointer in DNS RR Parser *(High)*

Retest Notes: Resolved by Stork, and verified by 7ASecurity.

The Stork server was identified as being vulnerable to a Denial-of-Service (DoS) condition caused by a nil pointer dereference in the DNS Resource Record (RR) parsing logic. A specially crafted string, when passed to the *NewRR* function, causes the underlying *dns.NewRR* function to return a nil resource record without raising an error. The Stork code fails to handle this nil case and subsequently attempts to dereference the nil pointer, triggering a panic that terminates the server process and causes a complete loss of service.

This issue was discovered through fuzz testing. The root cause lies in the incomplete

handling of return values from the call to *dns.NewRR(s)* within the *NewRR* function. While the code verifies whether an error is returned, it incorrectly assumes that a nil error always indicates a valid, non-nil resource record object. For certain malformed inputs, the *miekg/dns* library can return *(nil, nil)*, leading the subsequent logic in Stork *NewRR* function to operate on the uninitialized *rr* variable, resulting in a nil pointer dereference and server crash.

The provided stack trace confirms that a call to *isc.org/stork/appcfg/dnsconfig.NewRR* leads to a *"runtime error: invalid memory address or nil pointer dereference"*, demonstrating that the function does not correctly handle the edge case returned by the underlying dependency.

Affected File:
https://gitlab.isc.org/isc-projects/stork/-/blob/[...]/backend/appcfg/dnsconfig/rr.go#L32

Affected Code:
```
func NewRR(rrText string) (*RR, error) {
	rr, err := dns.NewRR(rrText)
	if err != nil {
		return nil, errors.Wrapf(err, "failed to parse RR: %s", rrText)
	}
	fields := strings.Fields(rr.String())
}
```

PoC:
The vulnerability was verified using a fuzzing harness that repeatedly invoked the *NewRR* function with generated input data. A specific malformed string input, representing a structurally invalid resource record, triggered a panic when *dns.NewRR* returned *(nil, nil)*. The following stack trace demonstrates the resulting crash:

Stack Trace:
```
testing.go:1693: panic: runtime error: invalid memory address or nil pointer
dereference goroutine 40 [running]: runtime/debug.Stack()
/Users/FuzzingUser/go/pkg/mod/golang.org/toolchain@v0.0.1-go1.24.9.darwin-arm64/src/run
time/debug/stack.go:26 +0xc4 testing.tRunner.func1()
/Users/FuzzingUser/go/pkg/mod/golang.org/toolchain@v0.0.1-go1.24.9.darwin-arm64/src/tes
ting/testing.go:1693 +0x21c panic({0x104ca8860?, 0x104f465d0?})
/Users/FuzzingUser/go/pkg/mod/golang.org/toolchain@v0.0.1-go1.24.9.darwin-arm64/src/run
time/panic.go:792 +0x124 isc.org/stork/appcfg/dnsconfig.NewRR({0x0, 0x0})
/Users/FuzzingUser/stork-audit/stork-v2-fuzzing/backend/appcfg/dnsconfig/rr.go:32
+0x12c isc.org/stork/appcfg/dnsconfig.FuzzDNSConfigRR.func1(0x140002541c0?, {0x0, 0x0})
/Users/FuzzingUser/stork-audit/stork-v2-fuzzing/backend/appcfg/dnsconfig/fuzz_test.go:1
45 +0x68 reflect.Value.call({0x104c9be80?, 0x104d02d10?, 0x1400025ee38?}, {0x104c0338c,
0x4}, {0x1400022b4a0, 0x2, 0x10495a6d0?})
```

It is recommended to update the *NewRR* function to include a *nil* check for the *rr* variable

immediately after the call to *dns.NewRR*. If the returned record is nil, the function should explicitly return an error, such as *errors.New("nil resource record returned by dns.NewRR")*. This ensures that the code never attempts to dereference a nil pointer, preventing unhandled panics and avoiding service disruption caused by invalid DNS input.

STO-01-012 WP6: DoS via Nil Map in Kea Config Merge *(High)*

Retest Notes: Resolved by Stork, and verified by 7ASecurity.

The Stork server was identified as being vulnerable to a Denial-of-Service (DoS) condition caused by a panic triggered when attempting to write to a nil map during the Kea configuration merge process. An attacker who provides a malformed or specially crafted Kea configuration can exploit this condition to terminate the server process, resulting in a complete loss of service.

This issue was confirmed through fuzz testing. The vulnerability originates in the *(*Config).Merge* method, which merges a source configuration into a destination configuration. The method retrieves the destination configuration (*destConfig*) but fails to verify whether it is nil before passing it to the recursive merge helper function.

If *destConfig* is *nil*, the *merge* function receives it as a nil map (*c1*). When the function attempts to add a key-value pair from the source configuration by executing *c1[k] = v2*, a panic occurs with the error message *"assignment to entry in nil map"*. This unhandled panic propagates upward and causes the Stork server process to crash.

Affected File:
https://gitlab.isc.org/isc-projects/stork/-/blob/[...]/backend/appcfg/kea/keaconfig.go#L550

Affected Code:
```
if v2 != nil {
        c1[k] = v2
}
```

PoC:
The vulnerability was confirmed using the Go fuzzer. A seed corpus entry supplied a configuration that caused a *nil* destination map to be passed to the merge logic, resulting in a panic. The following stack trace illustrates the crash condition:

Command:
```
❯ go test -fuzz=FuzzKeaConfig
```

Output:

```
fuzz: elapsed: 0s, gathering baseline coverage: 0/1884 completed
failure while testing seed corpus entry: FuzzKeaConfig/seed#35
fuzz: elapsed: 0s, gathering baseline coverage: 28/1884 completed
--- FAIL: FuzzKeaConfig (0.29s)
    --- FAIL: FuzzKeaConfig (0.00s)
        testing.go:1693: panic: assignment to entry in nil map
            goroutine 34 [running]:
            runtime/debug.Stack()

/Users/FuzzingUser/go/pkg/mod/golang.org/toolchain@v0.0.1-go1.24.9.darwin-arm64/src/run
time/debug/stack.go:26 +0xc4
            testing.tRunner.func1()

/Users/FuzzingUser/go/pkg/mod/golang.org/toolchain@v0.0.1-go1.24.9.darwin-arm64/src/tes
ting/testing.go:1693 +0x21c
            panic({0x100bb2840?, 0x100e93400?})

/Users/FuzzingUser/go/pkg/mod/golang.org/toolchain@v0.0.1-go1.24.9.darwin-arm64/src/run
time/panic.go:792 +0x124
            isc.org/stork/appcfg/kea.merge({0x100bb94a0, 0x0}, {0x100bb94a0,
0x140003aa4b0})

/Users/FuzzingUser/stork-audit/stork-v2-fuzzing/backend/appcfg/kea/keaconfig.go:551
+0x400
            isc.org/stork/appcfg/kea.(*Config).Merge(0x140003aa480, {0x100c1d380?,
0x140000a3bc0?})

/Users/FuzzingUser/stork-audit/stork-v2-fuzzing/backend/appcfg/kea/keaconfig.go:512
+0x108
            isc.org/stork/appcfg/kea.FuzzKeaConfig.func1(0x140000987d8?,
{0x14000381da1, 0x4})

/Users/FuzzingUser/stork-audit/stork-v2-fuzzing/backend/appcfg/kea/fuzz_test.go:299
+0x7a0
            reflect.Value.call({0x100ba3d40?, 0x100c190f0?, 0x14000098e38?},
{0x100aeda5f, 0x4}, {0x140003aa450, 0x2, 0x1007fb7c0?})
```

It is recommended to update the *(*Config).Merge* function to include a *nil* check for *destConfig* before invoking the *merge* helper. If *destConfig* is found to be *nil*, it should be initialized as an empty *RawConfig* map before being passed to the *merge* function. This ensures that the merge process operates on a valid map, preventing unhandled panics and avoiding complete service disruption.

Hardening Recommendations

This area of the report provides insight into less significant weaknesses that might assist adversaries in certain situations. Issues listed in this section often require another vulnerability to be exploited, need an uncommon level of access, exhibit minor risk potential on their own, and/or fail to follow information security best practices. Nevertheless, it is recommended to resolve as many of these items as possible to improve the overall security posture and protect users in edge-case scenarios.

STO-01-001 WP2 Enhanced Security Against MitM via TLS MinVersion *(Info)*

The Stork codebase currently supports TLS 1.2, but upgrading to TLS 1.3 is advised to enhance security. Although TLS 1.2 remains reliable and widely used, it is susceptible to certain cryptographic weaknesses and downgrade attacks[2]. Enforcing TLS 1.3[3] as the minimum version provides stronger security guarantees, broad compatibility, and proven stability after more than six years of availability. Exceptions may be made for legacy clients that require older TLS versions.

Affected File:
https://gitlab.isc.org/isc-projects/stork/[...]/restservice.go?ref_type=heads#L252

Affected Code:

```
func prepareTLS(httpServer *http.Server, s *RestAPISettings) error {
   var err error

   // Inspired by https://blog.bracebin.com/achieving-perfect-ssl-labs-score-with-go
   httpServer.TLSConfig = &tls.Config{
       // Only use curves which have assembly implementations
       // https://github.com/golang/go/tree/master/src/crypto/elliptic
       CurvePreferences: []tls.CurveID{tls.CurveP256},
       // Use modern tls mode
https://wiki.mozilla.org/Security/Server_Side_TLS#Modern_compatibility
       NextProtos: []string{"h2", "http/1.1"},
       //
https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet#Rule_-_Only_Supp
ort_Strong_Protocols
       MinVersion: tls.VersionTLS12,
       // These cipher suites support Forward Secrecy:
https://en.wikipedia.org/wiki/Forward_secrecy
       CipherSuites: []uint16{
           tls.TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,
           tls.TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,
           tls.TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,
```

[2] https://www.cloudflare.com/learning/ssl/why-use-tls-1.3/
[3] https://www.vertexcybersecurity.com.au/tls1-2-end-of-life/

```
            tls.TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,
            tls.TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305,
            tls.TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305,
        },
    }
[...]
```

It is recommended to configure all TLS instances to enforce *tls.VersionTLS13* as the minimum version to strengthen protection against potential downgrade and Man-in-the-Middle attacks. TLS 1.3 offers improved handshake security, reduced latency, and more robust cryptographic primitives. This configuration should be applied consistently across all Stork components that expose TLS endpoints. Where compatibility with legacy clients is necessary, exceptions should be explicitly documented and restricted to controlled environments.

STO-01-005 WP1: Possible Account Takeover via Password Policy *(Low)*

The Stork application was observed to allow users to set weak passwords. The only front-end validation enforced is a minimum length of eight characters (*minLength*). This weak password policy increases the likelihood of successful password guessing attacks against other accounts. Enforcing stronger password requirements, on both the client and server side, would reduce the risk of account compromise and strengthen overall authentication security.

Affected Files:
https://gitlab.isc.org/isc-projects/stork/[...]/users-page.component.ts?[...]
https://gitlab.isc.org/isc-projects/stork/[...]/password-change-page.component.ts?[...]

Affected Code:

```
passwordPattern: RegExp = /^[a-zA-Z0-9~`!@#$%^&*()_+\-=\[\]\\{}|;':",.\/<>?\s]+$/

constructor(
        private formBuilder: UntypedFormBuilder,
        private usersApi: UsersService,
        private msgSrv: MessageService,
        private auth: AuthService,
        private route: ActivatedRoute,
        private router: Router
) {}

ngOnInit() {
        this.passwordChangeForm = this.formBuilder.group(
            {
                oldPassword: ['', [Validators.required,
Validators.maxLength(this.maxInputLen)]],
                newPassword: [
```

```
                    '',
                    [Validators.required, Validators.minLength(8),
Validators.maxLength(this.maxInputLen)],
                ],
                confirmPassword: ['', [Validators.required,
Validators.maxLength(this.maxInputLen)]],
            }
        [...]
}
```

It is recommended to enforce stronger password requirements to improve account security. All user accounts should require passwords of at least twelve characters and a combination of character types, including uppercase letters, lowercase letters, numbers, and special characters. This enforcement should occur on both the server and client sides. These measures reduce the likelihood of account compromise through simple guessing or automated attacks and strengthen the overall security posture without introducing unnecessary complexity for users. For additional mitigation guidance, refer to the OWASP Authentication Cheat Sheet[4].

STO-01-006 WP1: Command Injection via Poisoned Agent Installer *(Medium)*

The Stork agent installer middleware was identified as dynamically generating shell scripts for remote agent installation, which makes it susceptible to command injection attacks. This issue stems from the insecure integration of the user-controlled HTTP *Host* header value directly into the shell script template generation process without adequate validation or sanitization. As a result, an attacker could inject arbitrary shell commands by crafting a malicious *Host* header and embedding its value into the script template used for agent installation, subsequently distributing the malicious script through other means.

To exploit this vulnerability, an attacker must control the HTTP *Host* header value. In the standard use case involving Stork installation via *wget*, the *Host* header is automatically and securely set by *wget* to match the *hostname* in the URL. However, an attacker can still manipulate the *Host* header using a command such as:

Command:

```
wget --header='Host: evil.com"; rm -rf /; echo "'
https://<server_address>/stork-install-agent.sh
```

The severity of this issue is reduced because it only enables the attacker to generate a malicious installation script, which still requires an additional distribution mechanism to achieve successful exploitation.

[4] https://cheatsheetseries.owasp.org/.../Authentication_Cheat_Sheet.html#...

Affected File:
https://gitlab.isc.org/isc-external/[...]/restservice/middleware.go?ref_type=heads#L170

Affected Code:

```
func agentInstallerMiddleware(next http.Handler, staticFilesDir string) http.Handler {
   // Agent installer as Bash script.
   const agentInstallerScript = `#!/bin/sh
set -e -x

rm -f /tmp/isc-stork-agent.{deb,rpm,apk}

{{ if .DebPath }}
if [ -e /etc/debian_version ]; then
   curl -o /tmp/isc-stork-agent.deb "{{.ServerAddress}}/{{.DebPath}}"
   DEBIAN_FRONTEND=noninteractive dpkg -i --force-confold /tmp/isc-stork-agent.deb
fi
{{ end }}
{{ if .ApkPath }}
if [ -e /etc/alpine-release ]; then
   wget -O /tmp/isc-stork-agent.apk "{{.ServerAddress}}/{{.ApkPath}}"
   apk add --no-cache --no-network /tmp/isc-stork-agent.apk
fi
{{ end }}
{{ if .RpmPath }}
if [ -e /etc/redhat-release ]; then
   curl -o /tmp/isc-stork-agent.rpm "{{.ServerAddress}}/{{.RpmPath}}"
   yum install -y /tmp/isc-stork-agent.rpm
fi
{{ end }}

systemctl daemon-reload
systemctl enable isc-stork-agent
systemctl restart isc-stork-agent
systemctl status isc-stork-agent

su stork-agent -s /bin/sh -c 'stork-agent register -u {{.ServerAddress}}'
`
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                if strings.HasPrefix(r.URL.Path, "/stork-install-agent.sh") {
                    pkgsRelativeDir := "assets/pkgs"
                    pkgsDir := path.Join(staticFilesDir, pkgsRelativeDir)
                    files, err := os.ReadDir(pkgsDir)
                    [...]

                    serverAddress := url.URL{
                        Scheme: scheme,
                        Host:   r.Host,
                    }

                    data := map[string]string{
```

```
            "ServerAddress": serverAddress.String(),
        }

        for extension, path := range packageFiles {
            key := strings.TrimLeft(extension, ".")
            key = strings.ToUpper(key[0:1]) + key[1:] + "Path"
            data[key] = path
        }

        t :=
template.Must(template.New("script").Parse(agentInstallerScript))
        err = t.Execute(w, data)
```

The *agentInstallerMiddleware* function is registered in the *GlobalMiddleware* chain and exposed as a REST API by the Stork server.

Affected File:
https://gitlab.isc.org/isc-projects/[...]/server/restservice/middleware.go?[...]

Affected Code:

```
func (r *RestAPI) GlobalMiddleware(handler http.Handler, staticFilesDir, baseURL
string, eventCenter eventcenter.EventCenter) http.Handler {
   // last handler is executed first for incoming request
   handler = fileServerMiddleware(handler, staticFilesDir)
   handler = agentInstallerMiddleware(handler, staticFilesDir)
   handler = sseMiddleware(handler, eventCenter)
   handler = metricsMiddleware(handler, r.MetricsCollector)
   handler = trimBaseURLMiddleware(handler, baseURL)
   handler = loggingMiddleware(handler)
   return handler
}
```

It is recommended to replace the *r.Host* value with a server-configured URL or to implement strict *Host* header validation against a whitelist. Client-supplied headers should never be trusted in security-critical operations such as generating executable scripts.

STO-01-007 WP1: Possible Weaknesses via Absent Security Headers *(Medium)*

It was found that the Stork web application does not set several important HTTP security headers. While this does not constitute a direct vulnerability, the absence of these protections may allow attackers to exploit weaknesses such as *TLS channel downgrades*[5], *Cross-Site Scripting (XSS)*[6], *Mime Sniffing*[7] and *Clickjacking*[8]. This was confirmed as follows in the rake demo instance:

Command:

```
curl -i --request GET --url http://localhost:8080/api/version
```

Output:

```
HTTP/1.1 200 OK
Server: nginx/1.29.2
Date: Tue, 14 Oct 2025 14:31:19 GMT
Content-Type: application/json
Content-Length: 46
Connection: keep-alive
Vary: Cookie

{"date":"2025-10-13 18:52","version":"2.3.1"}
```

It is recommended to configure the application and web server to include the following security headers in all responses, including error responses:

- *X-Frame-Options*: Determines if a page can be displayed within a frame, iframe, or object. While effective to protect from clickjacking attacks, a framable web page can facilitate many other attack scenarios[9]. SAMEORIGIN or DENY are appropriate values in most cases.
- It should be noted that some modern browsers prefer the *Content-Security-Policy frame-ancestors* directive, which offers similar protections. Deploying both headers can safeguard users across older and newer browsers.
- *X-Content-Type-Options*: Instructs the browser whether to allow MIME type sniffing for resources. Omitting this header is widely known to assist a specific attack scenario that manipulates the browser into rendering a resource as an HTML document, which ultimately incurs Cross-Site-Scripting (XSS).
- *Strict-Transport-Security (HSTS)*: This header is only effective when TLS/HTTPS is enabled. In deployments using HTTPS[10], its absence allows adversaries to downgrade HTTPS traffic to clear-text HTTP, hence facilitating MitM attacks

[5] https://en.wikipedia.org/wiki/Downgrade_attack
[6] https://owasp.org/www-community/attacks/xss/
[7] https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Headers_Cheat_Sheet.html#...
[8] https://owasp.org/www-community/attacks/Clickjacking
[9] https://cure53.de/xfo-clickjacking.pdf
[10] https://demo.stork.isc.org/

using widely available tools, like *sslstrip*[11]. It is advised to deploy HSTS as follows:

Strict-Transport-Security: max-age=31536000; includeSubDomains;

It is recommended to avoid the HSTS *preload* option due to its potential to cause DOS[12].

Security headers should be managed in a central location to ensure consistent application. This can be achieved through a load balancer, reverse proxy, or server configuration modules if changes to the application code are not feasible.

STO-01-008 WP1: Missing MFA Authentication *(Medium)*

The Stork web application supports internal and external (LDAP) authentication providers. However, internal authentication does not support multi-factor authentication (MFA). Although the application is generally expected to be hosted internally, its documentation considers the possibility of public exposure. In such cases, or when an attacker already has a foothold within the organization and attempts lateral[13] movement, strong authentication may serve as the only barrier preventing password-based attacks such as *password guessing*[14] or *password stuffing*[15].

It is recommended to implement MFA[16] for accounts using internal authentication mechanisms to prevent password-based attacks and to document the configuration of external authentication with MFA enabled. MFA should be mandatory for *super-admin* or privileged *break-glass* accounts. Ideally, the solution should integrate with an Identity Provider offering stronger security mechanisms than plain LDAP.

[11] https://moxie.org/software/sslstrip/
[12] https://www.tunetheweb.com/blog/dangerous-web-security-features/
[13] https://attack.mitre.org/tactics/TA0008/
[14] https://attack.mitre.org/techniques/T1110/001/
[15] https://attack.mitre.org/techniques/T1110/004/
[16] https://cheatsheetseries.owasp.org/cheatsheets/Multifactor_Authentication_Cheat_Sheet.html

WP3: Stork Supply Chain Implementation

Introduction and General Analysis

The *8th Annual State of the Software Supply Chain Report*, released in October 2022[17], reported an average yearly increase of 742% in software supply chain attacks since 2019. Some notable compromise examples include *Okta*[18], *Github*[19], *Magento*[20], *SolarWinds*[21], and *Codecov*[22], among many others. To mitigate this growing threat, Google released an End-to-End Framework for *Supply Chain Integrity* in June 2021[23], named *Supply-Chain Levels for Software Artifacts* (*SLSA*)[24].

This section evaluates the supply chain integrity of the Stork project using SLSA versions 0.1[25] and 1.0[26], which provide standardized methods for assessing software security and dependency management.

Current SLSA practices of Stork

Stork is an open-source management tool for ISC open-source software, integrating a graphical dashboard for Kea DHCPv4 and DHCPv6 servers. It provides filterable and sortable information, enabling single-screen monitoring of all servers. Users can access detailed health views, address plans (subnets and shared networks) with utilization data, and high-availability pair statuses.

From a SLSA perspective, Stork consists of two components: the Stork Server (*stork-server*) and the Stork Agent (*stork-agent*), both written in Go. The project uses the Rake build system and continuous integration via GitLab CI on AWS-hosted runners. It also leverages an internal build system for builds, unit tests, and system tests across multiple platforms.

[17] https://www.sonatype.com/press-releases/2022-software-supply-chain-report
[18] https://www.okta.com/blog/2022/03/updated-okta-statement-on-lapsus/
[19] https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/
[20] https://sansec.io/research/rekoobe-fishpig-magento
[21] https://www.techtarget.com/searchsecurity/ehandbook/SolarWinds-supply-chain-attack...
[22] https://blog.gitguardian.com/codecov-supply-chain-breach/
[23] https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html
[24] https://slsa.dev/spec/
[25] https://slsa.dev/spec/v0.1/
[26] https://slsa.dev/spec/v1.0/

Stork can be installed either through pre-built packages or by compiling from source. Pre-built packages are distributed via the Cloudsmith repository[27], while signed tarball releases are also available with instructions for signature verification[28].

These practices align with SLSA guidance, and the following sections analyze Stork implementation of specific SLSA requirements.

Source

Stork uses Git and GitLab for version control and codebase integrity. All contributions undergo review by trusted developers to ensure controlled and responsible repository access.

Build

The project employs the Rake build system orchestrated via GitLab CI/CD on AWS-hosted infrastructure, defined through a comprehensive *.gitlab-ci.yml*[29] pipeline configuration. The pipeline constructs native packages and runs automated tools, including Danger, Go Linter, Black, Flake, Pylint, Shell Linter, unit tests, and selected system tests. The repository is synchronized with GitHub for additional verification through Dependabot and CodeQL.

Additional security measures include sanity checks on tarballs and packages prior to release. Tarballs[30] are signed, and upcoming Stork 2.3.1 Git tags will also be signed before public release.

Releases are built in isolated environments with explicitly pinned[31] dependencies and automated testing. Both automated bots and maintainers manage dependency updates to maintain consistency and integrity.

[27] https://cloudsmith.io/~isc/repos/stork/packages/
[28] https://kb.isc.org/docs/aa-01225
[29] https://gitlab.isc.org/isc-projects/stork/-/blob/master/.gitlab-ci.yml?ref_type=heads
[30] https://downloads.isc.org/isc/stork/2.3.0/
[31] https://gitlab.isc.org/isc-projects/stork/-/blob/master/backend/go.mod?ref_type=heads#L6

Provenance

Stork builds are executed on GitLab, a platform supporting SLSA Provenance. Although Stork currently does not generate SLSA-compliant provenance, GitLab CI/CD supports this feature via the *RUNNER_GENERATE_ARTIFACTS_METADATA*[32] option. Enabling this option would allow automatic generation of signed provenance attestations that meet SLSA Level 1 requirements, including metadata such as commit SHA, pipeline ID, and build environment details.

SLSA v1.0 Framework Analysis

SLSA v1.0 defines four build levels describing software supply chain maturity:

- **Build L0: No guarantees** represent the lack of SLSA[33].
- **Build L1: Provenance exists**. The package has **provenance** showing how it was built. This can be used to prevent mistakes but is trivial to bypass or forge[34].
- **Build L2: Hosted build platform**. Builds run on a hosted platform that generates and signs the provenance[35].
- **Build L3: Hardened builds**. Builds run on a hardened build platform that offers strong tamper protection[36].

Based on documentation provided by the Stork team, 7ASecurity conducted a SLSA v1.0 assessment with the following results.

Producer requirements

Choose an Appropriate Build Platform

Stork uses GitLab as its build platform, which, when configured correctly, supports SLSA v1.0 Build Level 3 compliance.

Key configurations include:

1. **Containerized Build Environment:** Uses an immutable base container image ensuring consistent tooling and preventing infrastructure tampering.
    a. **File:** https://gitlab.isc.org/[...]/.gitlab-ci.yml?ref_type=heads#L1
    b. **Code:**

```
image: registry.gitlab.isc.org/isc-projects/stork/ci-base:9
```

[32] https://docs.gitlab.com/ci/runners/configure_runners/#artifact-provenance-metadata
[33] https://slsa.dev/spec/v1.0/levels#build-l0
[34] https://slsa.dev/spec/v1.0/levels#build-l1
[35] https://slsa.dev/spec/v1.0/levels#build-l2
[36] https://slsa.dev/spec/v1.0/levels#build-l3

2. **Managed Runner Infrastructure:** Utilizes AWS-hosted GitLab runners providing isolated and monitored environments.
    a. **File:** https://gitlab.isc.org/[...]/.gitlab-ci.yml?ref_type=heads#L476
    b. **Code:**
```
tags:
  - linux
  - aws
  - runner-manager
  - amd64
```

3. **Service Dependencies:** Declares all external build services, such as databases, ensuring consistent and traceable provisioning.
    a. **File:** https://gitlab.isc.org/[...]/.gitlab-ci.yml?ref_type=heads#L212
    b. **Code:**
```
services:
  - name: postgres:16-alpine  # Specific database version for testing
    alias: postgres           # Named service reference
```

At present, Stork achieves SLSA v1.0 Build Level 1, as it generates basic build metadata but does not yet produce structured, cryptographically signed provenance required for higher levels.

Follow a Consistent Build Process

Stork artifacts are produced through a scripted and consistent process within the GitLab CI/CD pipeline[37].

Key practices include:

1. **Structured Pipeline Definition:** Defines sequential CI/CD stages for predictable, automated builds.
    a. **File:** https://gitlab.isc.org/[...]/.gitlab-ci.yml?ref_type=heads#L4
    b. **Code:**
```
# stages order
stages:
  - build
  - checks
  - hooks
  - deploy
```

2. **Scripted Build Commands:** Standardized Rake-based build system ensuring automation and consistency.
    a. **File:** https://gitlab.isc.org/[...]/.gitlab-ci.yml?ref_type=heads#L274
    b. **Code:**

[37] https://gitlab.isc.org/isc-external/stork/-/blob/master/.gitlab-ci.yml

```
build_backend_amd64:
  extends: .base_build_debian
  script:
    - rake build:backend
```

3. **Comprehensive Build Matrix:** Defines explicit, documented processes for all components to ensure repeatable builds.
    a. **File:** https://gitlab.isc.org/[...]/.gitlab-ci.yml?ref_type=heads#L255
    b. **Code:**

```
script:
  - rake build:ui
  # and
  script:
  - rake build:doc
```

However, without signed provenance data, reproducibility cannot be verified across versions. Stork currently aligns with SLSA L1, which does not mandate signed provenance.

Build requirements

Distribute provenance

Stork achieves SLSA L1 through a structured GitLab pipeline that produces unsigned provenance metadata. To progress to L2 or L3, the team should enable *Artifact Provenance Metadata*[38] for structured provenance and distribute attestations via Cloudsmith.

Provenance Exists

The GitLab CI/CD pipeline provides basic provenance (commit SHA, pipeline ID, and environment details). This satisfies L1 but lacks signed and formatted attestations. Enabling SLSA-compliant provenance generation would elevate Stork to L2.

Positive existing practices include:

1. **Build Environment Tracking:** Immutable container images ensuring reproducibility.
    a. **File:** https://gitlab.isc.org/[...]/.gitlab-ci.yml?ref_type=heads#L1
    b. **Code:**

```
image: registry.gitlab.isc.org/isc-projects/stork/ci-base:9
```

[38] https://docs.gitlab.com/ci/runners/configure_runners/#artifact-provenance-metadata

2. **Comprehensive Quality Assurance Tracking:** Structured unit test, coverage, and artifact reports ensuring traceability
    a. **File:** https://gitlab.isc.org/[...]/.gitlab-ci.yml?ref_type=heads#L208-232
    b. **Code:**

```
unittest_backend:
  [...]
  coverage: '/Total coverage:\s+\d+.\d+%/'
  artifacts:
    reports:
      junit: backend/junit.xml
      coverage_report:
        coverage_format: cobertura
        path: backend/coverage.cobertura.xml
```

3. **Dependency and Cache Management:** Defined caching strategy for dependency tracking and performance optimization.
    a. **File:** https://gitlab.isc.org/[...]/.gitlab-ci.yml?ref_type=heads#L62-70
    b. **Code:**

```
cache:
    key: $CI_PROJECT_NAME-debian-$CI_COMMIT_REF_SLUG
    paths:
      - tools/
      - webui/node_modules/
      - /root/.cache/go-build
      - /var/lib/dpkg/info
    fallback_keys:
      - $CI_PROJECT_NAME-debian-$CI_DEFAULT_BRANCH
```

Provenance is Authentic and Unforgeable

Authenticity and non-forgeability can be achieved by enabling *Artifact Provenance Metadata* and signing with keys managed by the build platform. These features ensure provenance integrity and tamper resistance at SLSA Level 3.

Hosted and Isolated Builds

Stork builds are executed on AWS-hosted GitLab CI/CD infrastructure. According to the Stork team, CI jobs are executed in ephemeral, isolated environments (fresh containers and VMs per job) with restricted access.

SLSA v1.0 Assessment Results

The table below presents the results of Stork according to the Producer and Build platform requirements in the SLSA v1.0 Framework. The categories (source, build, provenance, and contents of provenance) are logically separated. Each row shows the SLSA level for each control, with green check marks indicating compliance and red boxes indicating the lack of evidence for compliance.

Implementer	Requirement		L1	L2	L3
Producer	Choose an appropriate build platform		✅	⛔	⛔
	Follow a consistent build process		✅	⛔	⛔
	Distribute provenance		✅	⛔	⛔
Build platform	Provenance generation	Exists	✅	⛔	⛔
		Authentic		⛔	⛔
		Unforgeable			⛔
	Isolation strength	Hosted		✅	✅
		Isolated			✅

SLSA v1.0 Conclusion

The assessment of Stork supply chain security confirms compliance with SLSA v1.0 Build Level 1. The build process is fully scripted and automated, with no manual intervention. All build steps are precisely defined in configuration files, such as *gitlab-ci.yaml*, and executed programmatically. Modifications to build definitions are tracked by the version control system, and package distribution is handled through *Cloudsmith*.

The current assessment of the Stork supply chain security demonstrates compliance with SLSA v1.0 Build L1, as the build process is entirely scripted and automated, eliminating manual intervention. All build steps are meticulously defined within configuration files, such as *gitlab-ci.yaml*, and are executed programmatically without human involvement. Furthermore, all modifications to the build definitions are meticulously tracked by the version control system. Finally, package distribution is managed through Cloudsmith.

However, higher levels require additional steps, including provenance generation, and cryptographic signing. This can be improved as follows:

- **SLSA v1.0 Build L2 Tamper resistance of the build service**: Integrate provenance attestation generation into the hosted build platform to automatically produce authenticated metadata for each build artifact. This requires:
  - Integrate the build platform by configuring GitLab CI/CD to produce SLSA provenance attestations in *in-toto* format.
  - Attestations must detail build metadata, including entry point, commands, source repository and commit SHA, platform/builder image, timestamp, and environment variables.
  - Associate each generated artifact (container images, binaries, packages) with its provenance attestation.
  - Store attestations in artifact registries (e.g., OCI, GitLab Package Registry) with artifacts or in dedicated provenance storage.
- **SLSA v1.0 Build L3 Hardened Builds with Signed Provenance:** Implement strong isolation guarantees and cryptographically signed attestations:
  - **Isolation requirements:**
    - Harden the build service to prevent tenant influence, ensuring that builds cannot affect the build platform or other builds.
    - Prohibit network access during build execution or implement strict egress filtering
  - **Provenance signing:**
    - Generate provenance attestations signed with cryptographic keys controlled by the build platform (not build requesters)
    - Implement key management using HSM, cloud KMS (AWS KMS, GCP KMS), or Sigstore
    - Use standard formats: in-toto attestations with Sigstore bundles
    - Include build platform identity in signatures to establish trust chain

This structured progression will enhance the integrity, authenticity, and traceability of the software supply chain while closing security gaps. While the Stork setup is sufficient for SLSA v1.0 Build Level 1, further upgrades are necessary to comply with SLSA v1.0 Build Level 2 and beyond, particularly in provenance generation and security measures.

SLSA v0.1 Framework

SLSA v0.1 outlines five levels of maturity for software supply chain security practices within a project, designated as "LX." In SLSA v1.0, these same maturity levels are referred to as "Build Level X," reflecting version-specific terminology.

- **L0: No guarantees.** This level represents the lack of any SLSA level.
- **L1:** The build process must be fully scripted/automated and generate provenance.
- **L2:** Requires using version control and a hosted build service that generates authenticated provenance.
- **L3:** The source and build platforms meet specific standards to guarantee the auditability of the source and the integrity of the provenance respectively.
- **L4:** Requires a two-person review of all changes and a hermetic, reproducible build process.

SLSA v0.1 Assessment Results

The following sections summarize the results of the software supply chain security implementation audit based on the SLSA v0.1 framework. Green check marks indicate that evidence of the noted requirement was found.

Requirement	L1	L2	L3	L4
Source - Version controlled		✅	✅	✅
Source - Verified history			✅	✅
Source - Retained indefinitely			✅	✅
Source - Two-person reviewed				✅
Build - Scripted build	✅	✅	✅	✅
Build - Build service		✅	✅	✅
Build - Build as code			✅	✅
Build - Ephemeral environment			✅	✅
Build - Isolated			✅	✅

Build - Parameterless				⛔
Build - Hermetic				⛔
Build - Reproducible				⛔
Provenance - Available	✅	⛔	⛔	⛔
Provenance - Authenticated		⛔	⛔	⛔
Provenance - Service generated		⛔	⛔	⛔
Provenance - Non-falsifiable			⛔	⛔
Provenance - Dependencies complete				⛔
Common - Security				⛔
Common - Access				⛔
Common - Superusers				⛔

SLSA v0.1 Level 2 and above are not met due to the absence of authenticated, service-generated provenance.

SLSA v0.1 Conclusion

The Stork supply chain security meets SLSA v0.1 Level 1, with infrastructure sufficient for certification. However, the absence of signed and formatted provenance represents a bottleneck. Addressing this gap will enable SLSA v0.1 Level 2+ compliance, meet regulatory expectations, and provide cryptographic proof of build integrity. Higher levels require additional steps, including provenance generation, and cryptographic signing:

- **SLSA v0.1 L2 Tamper resistance of the build service:**
  - Implement provenance signing using Sigstore, Cosign, or GitLab signing capabilities.
  - Adopt in-toto provenance format for standardized, machine-readable attestations.
  - Validate that version control and hosted build services are properly configured.
- **SLSA v0.1 Level 3 – Extra Resistance to Specific Threats:**

  - Verify parameterless build execution.
  - Ensure hardened build platform configuration.
  - Validate generation of non-falsifiable provenance.
- **SLSA v0.1 Level 4 – Highest Assurance and Trust:**
  - Document a two-person review process (for example, GitLab merge request approvals).
  - Confirm hermetic builds with pinned dependencies.
  - Validate reproducible build capabilities.
  - Ensure complete dependency tracking in provenance.

WP4: Stork Lightweight Threat Model

Introduction

Stork is an open-source application developed to manage critical network services through a centralized web-based graphical interface. It is primarily used to monitor, troubleshoot, and maintain configurations of Kea DHCP servers and to provide monitoring and limited control of BIND DNS servers.

Technically, the Stork system uses a client-server architecture with a PostgreSQL backend, a REST API, and primarily relies on gRPC with TLS between the server (*stork-server*) and agents (*stork-agent*). It leverages Kea extensions to safely propagate configuration changes. For advanced data monitoring and visualization, the software can integrate with Grafana and Prometheus.

By serving as a single centralized control point for critical network infrastructure services, Stork represents an attractive target for various attackers, as the data it collects and the network components it configures can be used to conduct sophisticated operations within compromised environments.

The threat model analysis in this document identifies security threats and vulnerabilities to enable early mitigation. The document, together with the related attack scenarios, establishes a baseline to encourage all team members to adopt a threat-led mindset towards everything designed and implemented, focusing on security from the outset to address risks before they evolve into exploitable vulnerabilities. A lightweight STRIDE-based approach[39] was applied using documentation, source code, existing threat models, research of underlying technologies, and client input to assess Stork.

This section classifies attack scenarios, outlines potential vulnerabilities, and proposes mitigations. The analysis focuses on Stork components, processed data, integrated external components, and briefly examines supply chain attack scenarios.

Relevant assets and threat actors

The following key assets were identified as significant for security:

- Internal CA infrastructure for issuing agent certificates
- Database credentials
- User credentials (read-only, admin, super admin)
- LDAP integration credentials
- BIND9 RNDC keys

[39] https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats#stride-model

- Kea control agent basic authentication credentials
- DNS zones and DHCP Leases
- Stork Agent Binaries

The following threat actors are considered relevant for the analysis:

- External attacker
- LAN attacker (for example, internal network)
- Advanced Persistent Threat (for example, hacking group, nation-state)

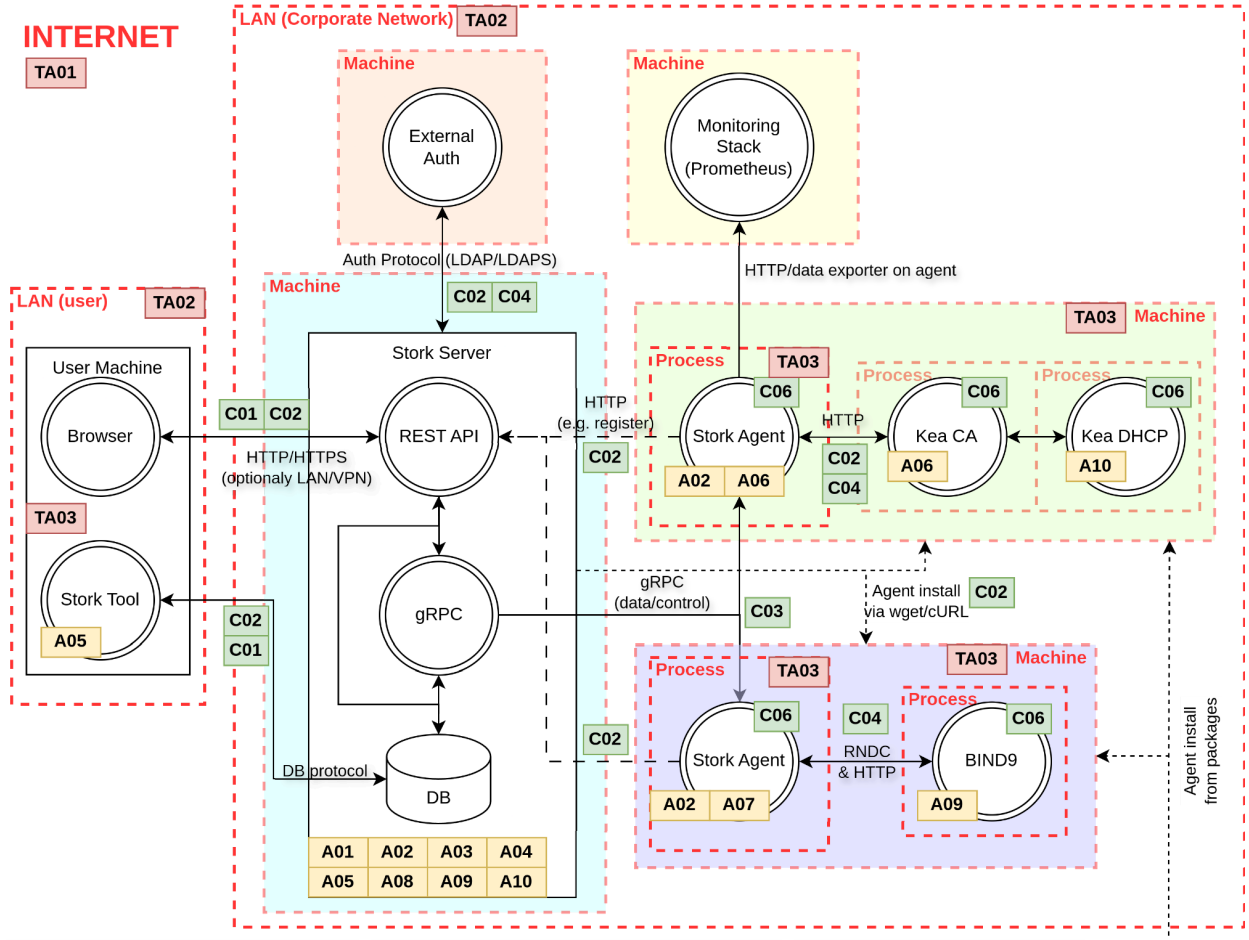
Attack surface

In threat modeling, the attack surface includes all potential entry points that an attacker may exploit to compromise a system, access or manipulate sensitive data, or disrupt application availability. Identifying the attack surface helps locate vulnerabilities and implement defenses to reduce risk. By analyzing threats and attack scenarios, organizations gain insight into potential techniques that could undermine system security.

Countermeasures

The following practices were identified based on available documentation and system information:

- Mutual TLS authentication between the server and agents over gRPC.
- Optional TLS support (for example, HTTPS or PostgreSQL) for administrators configuring the deployment
- Read-Only, Admin, and SuperAdmin roles implemented in the web interface
- Extensive documentation describing the use of encrypted protocols and OS-based permissions to protect the web application, daemons, and connections between the server, agent, Kea/BIND, and databases
- Agent registration confirmation
- Seamless agent registration using a privileged server registration token
- Basic LDAP integration through an optional hook mapping users and groups to privileges
- Signed artifacts available in the operating system package repository



Assets	
ID	Description
A01	CA Cretificate
A02	mTLS certificates
A03	LDAP Credentials
A04	Application Data (settings/users etc.)
A05	Database Credentials
A06	Kea Basic Auth Credentials
A07	RNDC keys
A08	Stork Agent Binaries
A09	DNS Zones
A10	DHCP Leases

Security Controls	
ID	Description
C01	User Credentials (username/password)
C02	optional TLS (e.g. HTTPS)
C03	mTLS (mutual authentication)
C04	Credentials (API key, keys, Basic Auth etc.)
C05	Artifacts Signatures
C06	OS-based permissions (user/service etc.)

Threat Actors	
ID	Description
TA01	External Attacker
TA02	LAN Attacker
TA03	Advanced Persistent Threat

Fig.: Data flow diagram involving the most important components and data flows

Threat 01: Compromised or Rogue Stork Agent

The Stork Agent functions as a trusted component within the managed infrastructure. It is responsible for data collection and command execution, serving as a communication channel between the Stork Server and Kea/BIND components. Compromise of a legitimate agent or registration of a rogue agent presents a significant concern due to potential unforeseen attack vectors exploitable during an advanced persistent threat scenario.

Attack Scenarios

- Lateral movement to Kea and BIND daemons by sending commands or reading sensitive resources such as configuration files, allowing stealthy reconfiguration of network components while concealing malicious entries from the Stork backend.
- Privilege escalation to root level through exploitation of misconfigurations or insufficient hardening of the Stork Agent service, deployed using systemd, supervisor, or other supported strategies.
- Malicious data transmitted to the Stork Server by a rogue agent, targeting the dashboard or internal components, such as spoofed hostnames, operating systems, or falsified Kea/BIND data.
- Denial of service through interception or blocking of commands from the Stork Server, achieved by cloning an agent on an attacker-controlled host and forcing the server to connect to the rogue instance.
- Server-side request forgery (SSRF) attempts triggered by malicious registration requests, establishing connections from the Stork Server to attacker-controlled hosts.
- Server registration token theft due to improper clearing from memory or insecure storage on the host, allowing seamless registration of malicious agents.
- Man-in-the-middle attacks during Stork Agent installation if the installation script is transferred over an unencrypted channel or lacks integrity verification.

Recommendations

- A comprehensive host review should be performed, focusing on hardening the Stork Agent service deployed via systemd or supervisor to prevent privilege escalation. This includes memory analysis to ensure registration tokens cannot be recovered.
- All commands issued by the Stork Agent to Kea/BIND should be auditable, supported by proper logging and monitoring sinks for all components. Anomaly detection templates should detect unauthorized commands originating outside the Stork Server.

- Data from the Stork Agent should be treated as untrusted, with rigorous validation covering rogue agent scenarios.
- Installation script integrity should be verified, and warnings should appear if unencrypted download methods are used. Package-based installation is preferred to mitigate man-in-the-middle risks.
- Mandatory HTTPS usage should be enforced, even in internal environments, to mitigate internal and advanced persistent threat risks.
- Automatic server registration token rotation should be implemented, with full auditability of all agents registered using each token, to identify and remove potentially malicious agents.
- On-demand generation of registration tokens should be enforced, ensuring tokens are not reusable once issued, thereby reducing exposure risk.

Threat 02: Supply Chain and Dependency Vulnerabilities

The security of the Stork software depends on the integrity of its development lifecycle and external dependencies. A compromise in the software supply chain, through either malicious code injection or exploitation of a vulnerable library, can introduce deeply embedded flaws. Such compromise may result in the inclusion of backdoors in official releases, rendering all deployments insecure from inception.

Attack Scenarios

- Compromise of the source code repository of any component used by Stork.
- Exploitation of the LDAP hook used for external authentication integration.
- Introduction of a backdoor via a compromised library used by the Stork Server, Agent, or other components.
- Credential leakage from the repository, allowing impersonation of legitimate developers and unauthorized commits.
- Use of vulnerable third-party libraries within Stork components (Server, Agent, or UI), exposing the system to known exploits.

Recommendations

- It is recommended to periodically scan all project dependencies for known vulnerabilities, including both backend and front-end components.
- Use the latest supported library versions to ensure timely application of vendor security patches. Pin and upgrade dependencies only after verifying all changes.
- Enforce a mandatory peer-review process requiring at least one approval before code merges and releases.
- Enforce commit signing at the repository level to prevent unauthorized or spoofed code submissions, and automatically reject unsigned commits.
- Periodically review artifact repositories to ensure strong security configurations, such as verified providers on Docker Hub for BIND[40] components or signed containers in dockerized deployments.
- Implement automated secret scanning to prevent exposure of sensitive data, and define clear procedures for secret rotation in case of an incident. Noted as Stork GL#2098[41].

[40] https://hub.docker.com/r/internetsystemsconsortium/bind9
[41] https://gitlab.isc.org/isc-projects/stork/-/issues/2098

Threat 03: Insecure Configuration and Backend Vulnerabilities

The overall security of the Stork platform depends on the secure configuration and deployment of its backend components. Environmental weaknesses such as improperly exposed APIs, unencrypted database connections, or weak credentials can enable direct system compromise. These issues may bypass application-level security controls, leading to the exposure of operational data and complete loss of platform integrity.

Attack Scenarios

- Argument injection vulnerability in the BIND 9 integration, which constructs shell commands (RNDC), allowing privilege escalation on the host system.
- Man-in-the-middle attack on an unencrypted connection between the Stork Server and PostgreSQL database, allowing credential theft or data tampering.
- Weak or default credentials for PostgreSQL or other components, enabling attackers with network access to communicate directly with APIs.
- Compromise of public documentation or demo sites, leading to reputational damage or dissemination of insecure configuration practices.
- Verbose debugging output (for example, during LDAP integration) that exposes sensitive information such as user passwords.
- Theft of Stork Tool credentials allowing direct database modification.
- Insecure database connections using Stork Tool, enabling man-in-the-middle attacks.

Recommendations

- Implement strict data validation for all data passed to integrated components.
- Generate strong, random credentials during installation for all environment components.
- Clearly mark demo installations or debugging modes as insecure to ensure users understand that such setups may log credentials or contain weak defaults.
- Harden demo environments to prevent reputational damage in the event of compromise.
- Provide secure deployment guidelines for all components, covering endpoint lists, supported protocols, and authentication methods. Integration of security scanning tools should be straightforward to enable early detection of insecure configurations or practices. Noted as Stork GL#2093[42].
- Establish robust logging and monitoring configurations to prevent unauthorized privileged access to the database or administrative accounts from untrusted sources or over insecure protocols.

[42] https://gitlab.isc.org/isc-projects/stork/-/issues/2093

- Configure Stork Tool to warn users when insecure connections are detected and ensure that all actions are logged. Noted as Stork GL#2093[43].
- Document integration with centralized logging solutions to enable complete auditability and ensure non-repudiation. Noted as Stork GL#2093[44].

Threat 04: Web Dashboard Vulnerabilities and Data Exposure

The Stork Dashboard functions as the central administrative interface of the platform, making its security critical for protecting managed services. Vulnerabilities in the web application or its backend APIs can enable attackers to steal session data, escalate privileges, or compromise the underlying server. Such exploitation can lead to exposure of sensitive configuration information and facilitate lateral movement within the network.

Attack Scenarios

- Injection of malicious content into the dashboard through compromised Stork Agent components (for example, OS name, hostname, logs, or raw/JSON configuration data), potentially leading to stored cross-site scripting (XSS), Prototype Pollution, or other web-based attacks.
- Password-guessing[45] or credential-stuffing[46] attacks against privileged accounts, enabling access to sensitive data such as Kea configurations that may contain database credentials usable for lateral movement or network reconfiguration[47][48].
- Internal phishing campaigns targeting Stork Dashboard users to harvest LDAP credentials within the organization.
- Compromise of low-privileged user accounts through weak passwords, allowing unauthorized access to DNS data and passive reconnaissance.

Recommendations

- Conduct periodic web security assessments, especially before major feature releases.
- Include gRPC fuzz testing of Stork Agent interactions to validate data integrity and prevent injection flaws.
- Mask credentials and other sensitive data displayed in the web interface for all user roles, including *super-admin*.
- Integrate enterprise-grade identity providers supporting multi-factor authentication to reduce the likelihood of password-based attacks and limit

[43] https://gitlab.isc.org/isc-projects/stork/-/issues/2093
[44] https://gitlab.isc.org/isc-projects/stork/-/issues/2093
[45] https://attack.mitre.org/techniques/T1110/001/
[46] https://attack.mitre.org/techniques/T1110/004/
[47] https://attack.mitre.org/tactics/TA0008/
[48] https://attack.mitre.org/techniques/T1557/003/

phishing impact. Suitable examples include Keycloak, FreeIPA, Okta, and Entra ID. Noted as Stork GL#1373[49].

[49] https://gitlab.isc.org/isc-projects/stork/-/issues/1373

Conclusion

Despite the findings encountered in this exercise, the Stork solution defended itself well against a broad range of attack vectors. The platform will become increasingly difficult to attack as additional cycles of security testing and subsequent hardening continue.

The Stork components in scope provided a number of positive impressions during this assignment that must be mentioned here:

- Overall, the solution was found to be robust against many traditional web application security attack vectors. For example, no *Command Injection*, *Cross-Site Request Forgery (CSRF)*, *Local File Inclusion (LFI)* or *Remote Code Execution (RCE)* issues could be identified during this assignment.
- HTML Form submissions and API endpoints were both found to be safely protected against CSRF.
- The source code demonstrated clear structure and compliance with secure development best practices.
- The testing environments were organized and supported effective verification activities.
- Robust input validation routines were consistently implemented across modules, indicating a mature security posture.
- Angular sanitization mechanisms reduced the risk of HTML or script injection.
- The modular architecture and scoped component design limit the potential impact of crashes or malformed input to isolated subsystems.
- Access control was found to be reliable, and no unauthorized data exposure was observed. The codebase reflected proactive handling of external input and adherence to secure coding principles throughout the development process.
- The application was not found to reveal any sensitive data or cookies to third party websites.

The security of the Stork solution will improve with a focus on the following areas:

- **Nil Pointer and Error Handling:** Nil checks must be added for all variables and configuration structures that may trigger runtime panics, particularly within DNS record parsing and configuration merging functions, to improve reliability (STO-01-011, STO-01-012).
- **Input Validation and Data Handling:** The agents must be hardened against malformed data. The issue identified in the Kea statistics parser (STO-01-002) demonstrates a risk of monitoring interruption. All regular expression matches and external inputs must be validated before use to prevent runtime crashes and service disruption.
- **Parser Stability:** The PowerDNS parser must enforce a maximum field count and line length limits to prevent excessive memory allocation and

denial-of-service conditions caused by malformed configuration input (STO-01-009, STO-01-010).

- **HTML Rendering in the User Interface:** All external data must be treated as untrusted in the user interface. The *[innerHTML]* bindings must be replaced with safe text interpolation to fully prevent HTML injection via daemon version strings (STO-01-004).
- **SQL Query Construction:** Unsafe dynamic query construction must be eliminated. The latent SQL injection identified in the stork-tool (STO-01-003) indicates the need to validate all user-supplied database identifiers against a restrictive allow-list pattern (for example, allowing only letters, numbers, and underscores) before inclusion in database statements.
- **Command Injection:** Although input filtering across the source code is effective, the agent installer script was vulnerable to command injection, which could compromise the agent installer process. Input handling within this script must be refactored to remove command injection risks (STO-01-006).
- **Transport Layer Security:** To mitigate Man-in-the-Middle (MiTM) scenarios, stronger transport security must be enforced by configuring all components to use TLS with a defined minimum version (STO-01-001).
- **Security headers:** The absence of HTTP security headers in the Stork web application must be addressed. Headers such as *X-Frame-Options*, *X-Content-Type-Options*, and *Strict-Transport-Security* should be implemented to improve client-side protection (STO-01-007).
- **Password Policy:** A stronger password policy must be enforced on both client and server sides, requiring longer and more complex passwords to improve protection against brute-force and password-guessing attacks (STO-01-005). Noted as Stork GL#2124[50].
- **Authentication Hardening:** Multi-Factor Authentication should be implemented for privileged accounts, including administrative and break-glass access, to reduce the risk of unauthorized access (STO-01-008). Noted as Stork GL#1373[51].

It is advised to address all issues identified in this report, including informational and low severity tickets where possible. This will not just strengthen the security posture of the application significantly, but also reduce the number of tickets in future audits.

Once all issues in this report are addressed and verified, a more thorough review, ideally including another source code audit, is highly recommended to ensure adequate security coverage of the platform. This provides auditors with an edge over possible malicious adversaries that do not have significant time or budget constraints.

[50] https://gitlab.isc.org/isc-projects/stork/-/issues/2124
[51] https://gitlab.isc.org/isc-projects/stork/-/issues/1373

Please note that future audits should ideally allow for a greater budget so that test teams are able to deep dive into more complex attack scenarios. Some examples of this could be third party integrations, complex features that require to exercise all the application logic for full visibility, authentication flows, challenge-response mechanisms implemented, subtle vulnerabilities, logic bugs and complex vulnerabilities derived from the inner workings of dependencies in the context of the application. Additionally, the scope could perhaps be extended to include other internet-facing Stork resources.

It is suggested to test the application regularly, at least once a year or when substantial changes are going to be deployed, to make sure new features do not introduce undesired security vulnerabilities. This proven strategy will reduce the number of security issues consistently and make the application highly resilient against online attacks over time.

7ASecurity would like to take this opportunity to sincerely thank Andrei Pavel, Marcin Siodelski, Piotrek Zadroga, Slawek Figiel, Tomek Mrugalski, Vicky Risk, Wlodzimierz Wencel, Yavor Peev and the rest of the Stork team, for their exemplary assistance and support throughout this audit. Last but not least, appreciation must be extended to the *Open Source Technology Improvement Fund (OSTIF)* for facilitating and managing this project, and the *Internet Systems Consortium (ISC)* for funding it.

License and Legal Notice

This report is licensed under the *Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)*[52] license.
You are free to:

- **Share** – copy and redistribute the material in any medium or format
- **Adapt** – remix, transform, and build upon the material for any purpose, even commercially

Under the following terms:

- **Attribution** – You must give appropriate credit to 7ASecurity, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests 7ASecurity endorses you or your use.

- **ShareAlike** – If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

Exceptions and Restrictions:

- **Trademarks and Logos**: The 7ASecurity name, logo, and visual identity elements (such as custom fonts or design marks) are not licensed under CC BY-SA 4.0 and may not be used without explicit written permission.

- **Third-party Content:** Any third-party content (e.g., open source project logos, screenshots, excerpts) included in this report remains under its respective copyright and licensing terms.

- **No Endorsement:** Use of this report does not imply endorsement by 7ASecurity of any derivative works, use cases, or conclusions drawn from the material.

Disclaimer: This report is provided for informational purposes only and reflects the state of the target project at the time of testing. No warranties are provided. Use at your own risk.

[52] https://creativecommons.org/licenses/by-sa/4.0/