



Code Security Review Report

Internal - Newtonsoft.Json

Client	Internal (Sample Report)
Application Name	Newtonsoft.Json (v12.0.3)
Classification	Unclassified
Review date	2026-09-18
Report Version	v1
Report Author(s)	Intuitive IT

Purpose	3
Executive summary	3
Scope & methodology	4
Risk rating methodology	4
Summary of findings	5
Detailed findings	6
HARD-1 Unbounded JSON nesting depth causes uncatchable stack-overflow DoS	6
HARD-2 TypeNameHandling deserialization RCE risk when misused by consumers	6
SEC-1 Strong-name key pair committed to repository (Dynamic.snk)	7
Positive observations	8
Recommendations & next steps	8
Appendix	8
Assumptions	8

Purpose

This document is a security review of the Newtonsoft.Json open-source .NET library, version 12.0.3, prepared by Intuitive IT for Example Report. It records the findings of a source-code and git-history review, and provides prioritized recommendations.

Executive summary

Newtonsoft.Json is a widely-used, mature open-source JSON library, not an internally hosted application — so most of the usual review areas (hosting, data storage, authentication, network exposure) don't apply here. The review focused on the two areas that do apply to a library like this: how safely it parses untrusted input by default, and whether any secrets or credentials were ever committed to its history.

The library is generally well-engineered from a security standpoint, with safe-by-default settings and no secrets found anywhere in its history. Two design-level weaknesses were identified that matter specifically for teams consuming this library version: the library will happily parse extremely deeply-nested JSON without any built-in limit, which can crash a consuming application; and one of its optional features, if turned on carelessly, is a well-known route to remote code execution in the consuming application (not in this library itself). Neither is a flaw an attacker can exploit in the library standing alone — both depend on how an application that uses this library is configured.

Findings at a glance:

Severity	Count
 Critical	0
 High	1
 Medium	1
 Low	1
 Informational	0

Most important next step: Any team consuming Newtonsoft.Json 12.0.3 should upgrade to 13.0.1 or later, which adds a depth limit to LINQ-to-JSON parsing (HARD-1), and should audit their own code for use of `TypeNameHandling` (HARD-2).

Scope & methodology

In scope: Static source code and full git commit history of the Newtonsoft.Json repository at git tag [12.0.3](#).

Out of scope: Live systems — none exist for this project. There is no hosting, deployment, datastore, authentication surface, or third-party runtime integration to review, because this is a standalone library, not a deployed application. Consuming applications that use this library are also out of scope; findings here describe risks that depend on how such an application configures the library.

Access & limitations: Code and full git history were available. No live-system access was applicable. One item — whether GitHub branch protection on the source repository requires CI checks to pass before merge — could not be confirmed from the static source and was descope for this review at the operator's direction.

Methodology: This review followed Intuitive IT's standard application review methodology, adapted for a library with no live/hosted component — covering architecture, secrets management, hardening, and code quality. Data storage, authentication, network exposure, audit logging, and third-party integration review were assessed as not applicable (see Appendix).

Risk rating methodology

Findings are rated on a qualitative scale based on the realistic likelihood of exploitation and the business impact if exploited — not on how alarming a vulnerability class sounds in the abstract. For each Critical, High, or Medium finding, the detailed write-up below also states the likelihood and blast radius that justify its rating.

Severity	Definition
 Critical	Directly and immediately exploitable by an external, unauthenticated party to access, modify, or exfiltrate sensitive data, or to take over the application/infrastructure.
 High	Exploitable by an authenticated user beyond their intended privilege, or by an external party under realistic conditions (e.g. a leaked-but-not-yet-abused credential, a missing server-side check reachable from the client).
 Medium	A real weakness that requires a specific precondition to exploit (insider access, chaining with another issue, a narrow

Severity	Definition
	window of opportunity), or that materially weakens defense-in-depth without being independently exploitable today.
● Low	A deviation from best practice with limited realistic impact on its own — hardening/hygiene work.
● Informational	Not a vulnerability. Worth recording: a design tradeoff, an out-of-scope item, an unverified/assumption-based item, or context useful for future reviews.

Positive observations (see below) are genuine strengths, not findings, and are not rated on this scale.

Summary of findings

ID	Title	Severity	Area	Owner
HARD-1	Unbounded JSON nesting depth causes uncatchable stack-overflow DoS (CVE-2024-21907)	● High	Hardening	Code change (consumer: upgrade dependency)
HARD-2	TypeNameHandling deserialization RCE risk when misused by consumers	● Medium	Hardening	Process decision (consumer: secure configuration)
SEC-1	Strong-name key pair committed to repository (Dynamic.snk)	● Low	Secret management	Code change

Detailed findings

HARD-1 Unbounded JSON nesting depth causes uncatchable stack-overflow DoS

Severity:  High

Tracked as: CVE-2024-21907 / GHSA-5crp-9r3c-p9vr

What this means: An attacker who can send arbitrarily-structured JSON to any application using this library version can crash that application's entire process with a single request, with no login required.

Details: JSON parsing in this library is recursive-descent, so deeply nested input (e.g. thousands of nested arrays) grows the call stack proportionally. `JsonReader.MaxDepth` exists as an opt-in limit, but it is `null` (unlimited) by default and must be explicitly set by the consuming application. Worse, the LINQ-to-JSON entry points (`JToken.Parse`, `JObject.Parse` — commonly used to parse arbitrary/untrusted JSON) have no depth-limiting mechanism available at all in this version; `JsonLoadSettings` carries no `MaxDepth` property. Once the recursion exhausts the thread stack, .NET raises a `StackOverflowException`, which cannot be caught by application code and terminates the process outright. This is assigned CVE-2024-21907 (GHSA-5crp-9r3c-p9vr), fixed upstream in Newtonsoft.Json 13.0.1.

Evidence: `Src/Newtonsoft.Json/JsonReader.cs` (nullable `MaxDepth`, defaults unset); `Src/Newtonsoft.Json/Linq/JsonLoadSettings.cs` (no `MaxDepth` property present in this version).

Recommendation: Upgrade the dependency to 13.0.1 or later. If an immediate upgrade isn't possible, have consuming applications explicitly set a conservative `JsonSerializerSettings.MaxDepth` (e.g. 64) on every deserialization call site, and avoid `JToken.Parse/JObject.Parse` on untrusted input until upgraded, since no depth control exists for that path in this version.

HARD-2 `TypeNameHandling` deserialization RCE risk when misused by consumers

Severity:  Medium

What this means: This library ships an optional feature that, if a development team turns it on without also adding a specific safeguard, can let an attacker who controls input JSON make the application run arbitrary code.

Details: `JsonSerializerSettings.TypeNameHandling` defaults safely to `None`, and the library's own documentation explicitly warns that any other value should be paired with a

custom, restrictive `SerializationBinder` when deserializing untrusted input. However, `DefaultSerializationBinder` — the binder used unless a consumer supplies their own — performs no type allowlisting whatsoever: it will resolve and load any type reachable by name and assembly. If a consuming application sets `TypeNameHandling` to `Auto`, `All`, or `Objects` and deserializes attacker-supplied JSON without a custom binder, an attacker can specify arbitrary `$type` values and, using known .NET deserialization gadget chains, achieve remote code execution. This is a long-documented risk pattern for this library across the .NET ecosystem, not a defect unique to 12.0.3, but it remains present in this version exactly as described.

Evidence: `Src/Newtonsoft.Json/JsonSerializerSettings.cs` (default `TypeNameHandling.None`, XML-doc warning);
`Src/Newtonsoft.Json/Serialization/DefaultSerializationBinder.cs` (no allowlist logic).

Recommendation: This is not a code defect to patch in the library, but a configuration risk to manage in any application consuming it: audit all uses of `TypeNameHandling` in downstream applications; where non-`None` values are required, pair them with a custom `SerializationBinder` that allowlists only the specific types expected, and never enable non-`None` `TypeNameHandling` on JSON originating from an untrusted party without one.

SEC-1 Strong-name key pair committed to repository (`Dynamic.snk`)

Severity:  Low

What this means: A private cryptographic key file is stored in the project's source history. Its practical impact is minimal, both because of what it's used for and because it turns out not to be used by anything in the current build.

Details: `Src/Newtonsoft.Json/Dynamic.snk` is committed to the repository and is consistent in size with a full RSA key pair (not a public-key-only `.snk`). `AssemblyInfo.cs` declares an `InternalsVisibleTo("Newtonsoft.Json.Dynamic, PublicKey=...")` entry that this key would correspond to, but no `.csproj/.props/.targets/.sln` file in this checkout references `Dynamic.snk`, and no `Newtonsoft.Json.Dynamic` project exists in the current tree to be signed with it — the file dates to 2009 and is leftover from an old companion assembly that no longer exists at this version. It is not used to sign the shipped NuGet package either — that key is supplied separately at CI time via a secured pipeline variable (`azure-pipelines.yml`) and is never committed. Because the key isn't wired into any current build, removing it would not affect a contributor's ability to build the library or run its test suite today.

Evidence: `Src/Newtonsoft.Json/Dynamic.snk` (596 bytes, added 2009);
`Src/Newtonsoft.Json/Properties/AssemblyInfo.cs`
(`InternalsVisibleTo("Newtonsoft.Json.Dynamic, ...")` entry with no corresponding project in this tree); repo-wide search of build files (`.csproj/.props/.targets/.sln`) found zero references to any `.snk` file;

`azure-pipelines.yml` (separate, non-committed package-signing key sourced from a secured variable).

Recommendation: Optional hygiene improvement: since this key is dead/unused in the current build, it can simply be deleted rather than regenerated — there is no working contributor workflow that depends on it.

Positive observations

- A full search of the entire git history found no committed secrets, credentials, private keys (other than the low-impact `Dynamic.snk` noted above), or API tokens.
 - The library's actual package-signing key is handled correctly: supplied only at CI time from a secured pipeline variable, never committed to source.
 - `TypeNameHandling` defaults safely to `None`, and the risk of enabling it is clearly documented in the library's own XML doc comments — a genuine secure-by-default design choice.
 - The core library has zero third-party runtime NuGet dependencies (only build-time analyzers and .NET base-class-library compatibility shims), eliminating an entire class of transitive dependency risk.
 - CI (Azure Pipelines and AppVeyor) runs the full NUnit/VSTest suite on every build.
-

Recommendations & next steps

1. Upgrade consuming applications from Newtonsoft.Json 12.0.3 to 13.0.1 or later to gain depth-limiting on LINQ-to-JSON parsing (HARD-1).
 2. Audit any consuming application for `TypeNameHandling` usage and ensure a restrictive custom `SerializationBinder` is in place wherever it's enabled on untrusted input (HARD-2).
 3. Optionally, delete the unused `Dynamic.snk` file (SEC-1).
-

Appendix

Assumptions

- Newtonsoft.Json is a standalone library with no hosting, deployment, or live infrastructure of its own; sections of the standard checklist that presuppose a deployed application were marked not applicable rather than force-fitted.

- Whether GitHub branch protection on the source repository requires CI checks to pass before merge was not verified; this was descope'd for this review at the operator's direction rather than left as an open unverified item.