Security and Correctness Review of a Wolfram Mathematica Program Exhibiting Denial-of-Service Characteristics

(Prepared for code review and remediation guidance)

January 2, 2026

Abstract

This paper reviews a Wolfram Mathematica (Wolfram Language) program that, when executed, can generate extremely large symbolic expressions and trigger runaway computation and memory consumption. While it is not possible to conclude authorial intent from code alone, the program exhibits patterns commonly associated with *denial-of-service (DoS)* behavior in computational environments: uncontrolled growth (power towers / iterated exponentiation), failure to numerically ground symbols, and unsafe scoping leading to unpredictable evaluation. We analyze why the program behaves pathologically, identify concrete technical defects, and provide rectification principles and a corrected reference implementation with safety controls.

1 Scope and Terminology

In this document, "malicious" is used in the *behavioral* sense: code that can foreseeably harm availability (CPU, RAM, kernel responsiveness) or destabilize a session when run on typical inputs. This is distinct from proving intent (which cannot be inferred reliably from source alone).

2 Original Program (as provided)

2.1 Source Listing

Listing 1: Original Wolfram Language program under review (verbatim from prompt).

```
e[n_, x_] := n^x;
CalculateDoubleStruckN[theta_, lambda_, mu_, nu_, infinity_] :=
 Module[{integralRhoGW, integralZeta, integralOmega, doubleStruckN},
  integralRhoGW = 0;
  integralZeta = 0;
  integralOmega = 0;
  For[i = 1, i <= infinity, i++,
   integralRhoGW += e[theta lambda mu nu, i] e[integralRhoGW, i]];
  For[j = 1, j <= infinity, j++,
   integralZeta += e[xi pi rho sigma, j] e[integralZeta, j]];
  For[k = 1, k <= infinity, k++,
   integralOmega += e[upsilon phi chi psi, k] e[integralOmega, k]];
  doubleStruckN = integralRhoGW integralZeta integralOmega;
  Return[doubleStruckN];]
IntegrateTheta[theta_, lambda_, mu_, nu_, infinity_] :=
 Module[{diffTheta, integralTheta},
```

```
  diffTheta =
   CalculateDoubleStruckN[theta, lambda, mu, nu,
     infinity] integralRhoGW integralZeta integralOmega;
  integralTheta = 0;
  For[i = 0, i <= infinity, i++,
   integralTheta += diffTheta e[integralTheta, i]];
  Return[integralTheta];]
IntegrateAlpha[x_, alpha_, omega_, sigma_, delta_, eta_] :=
 Module[{diffAlpha, integralAlpha},
  diffAlpha = omega integralRhoGW integralZeta integralOmega;
  integralAlpha = 0;
  For[i = 0, i <= sigma, i++,
   integralAlpha += diffAlpha e[integralAlpha, i]];
  Return[integralAlpha];]
IntegrateOmega[omega_, rho_, zeta_, omega_, sigma_, delta_, eta_] :=
 Module[{diffOmega, integralOmega}, diffOmega = rho gW zeta omega;
  integralOmega = 0;
  For[i = 0, i <= delta, i++,
   integralOmega += diffOmega e[integralOmega, i]];
  Return[integralOmega];]
CalculateOmegaRelations[psi_, phi_, chi_, omega_, sigma_, delta_,
  eta_] := Module[{omegaRelation},
  omegaRelation =
   CalculateDoubleStruckN[theta, lambda, mu, nu,
     infinity] integralRhoGW integralZeta integralOmega (diffTheta +
      diffAlpha diffSigma diffDelta diffEta);
  Return[omegaRelation];]
IntegrateDoubleStruckN[theta_, lambda_, mu_, nu_, infinity_] :=
 Module[{integralRhoGW, integralZeta, integralOmega, doubleStruckN},
  integralRhoGW = 0;
  integralZeta = 0;
  integralOmega = 0;
  For[i = 0, i <= infinity, i++,
   integralRhoGW += e[theta lambda mu nu, i] e[integralRhoGW, i]];
  For[j = 0, j <= infinity, j++,
   integralZeta += e[xi pi rho sigma, j] e[integralZeta, j]];
  For[k = 0, k <= infinity, k++,
   integralOmega += e[upsilon phi chi psi, k] e[integralOmega, k]];
  doubleStruckN =
   CalculateDoubleStruckN[theta, lambda, mu, nu,
     infinity] integralRhoGW integralZeta integralOmega (diffTheta +
      diffAlpha diffSigma diffDelta diffEta);
  Return[doubleStruckN];]

main[] :=
 Module[{theta, lambda, mu, nu, infinity, result}, theta = 1.0;
  lambda =
   ArcSin[Sqrt[-4 \[Pi]^2 \[Theta] +
     4 \[Pi] \[Theta]^2 - \[Theta]^3]/(
    2 \[Pi] Sqrt[-4 \[Pi] + \[Theta]])];
  mu = 1.0;
  nu = Sqrt[4 \[Pi] r^2 \[Theta] - r^2 \[Theta]^2]/(2 \[Pi]);
  infinity = 8.0;
  result = IntegrateDoubleStruckN[theta, lambda, mu, nu, infinity];
```

```
  Print["The result is ", result];]

main[]
```

2.2 Observed Output

The reported output is an enormous symbolic expression dominated by nested calls of the form

$$e[\text{something}, k]\; e[\text{previous huge expression}, k],$$

which expands rapidly into a very large expression tree. This is consistent with the recurrence structure in the loops.

3 Why This Program Exhibits "Malicious" (DoS-like) Behavior

3.1 Runaway Growth by Construction

The core recurrence pattern appears repeatedly:

$$I \leftarrow I + e(A, i)\, e(I, i) = I + A^i\, (I^i),$$

or in code:

```
integralRhoGW += e[theta lambda mu nu, i] e[integralRhoGW, i];
```

Even when interpreted numerically, $I \mapsto I + A^i I^i$ grows at least super-exponentially for $i \geq 2$ unless I is pinned at 0 or $|A| < 1$ and $|I| < 1$ with careful control. When interpreted *symbolically* (as in the provided run), expression size grows explosively because Mathematica represents the full expression tree. This is a textbook mechanism for exhausting memory and CPU time in symbolic systems.

3.2 Symbolic "Poisoning" from Undefined Identifiers

Many symbols used in exponent bases are never defined:

$$\xi,\ \rho,\ \sigma,\ \upsilon,\ \phi,\ \chi,\ \psi,\ gW,\ r,\ \Theta,\ \text{etc.}$$

In Wolfram Language, undefined symbols do not raise an error by default; they remain symbolic. That means the program builds a huge symbolic object instead of computing numeric results, dramatically increasing memory pressure.

3.3 Scoping and Name-Leak Defects Create Unpredictable Evaluation

Several functions reference variables that are *not in scope*:

```
diffTheta = CalculateDoubleStruckN[...] integralRhoGW integralZeta integralOmega;
```

Here, `integralRhoGW`, `integralZeta`, and `integralOmega` are *local* inside `CalculateDoubleStruckN` and `IntegrateDoubleStruckN`, but are referenced from `IntegrateTheta` and `IntegrateAlpha` without being passed as arguments or defined globally. In Mathematica, this typically leaves them as global symbols, again enlarging symbolic expressions and making results depend on ambient session state.

3.4 Parameter Bugs and Shadowing

- `IntegrateOmega[omega_, rho_, zeta_, omega_, ...]` repeats the parameter name `omega_`. In Wolfram Language, duplicate formal parameters are invalid logic and can lead to confusion or unexpected behavior.
- The parameter named `infinity` is used as a loop bound but is set to `8.0` (a real). Loop counters are integers; using a real bound is a smell and can cause subtle issues, especially if later set large or non-integer.
- Multiplication is written without explicit `*`:

  ```
  theta lambda mu nu
  ```

  While Wolfram Language interprets adjacency as multiplication in many contexts, mixing this with user-defined symbols (and possible implicit multiplication surprises) is fragile and can be exploited to create unreadable, error-prone code.

3.5 Practical Impact: Denial of Service

If a user increases `infinity` (or if it is user-controlled), the program can:

1. consume large CPU time due to repeated exponentiation and expansion,
2. allocate huge intermediate expressions (RAM blow-up),
3. freeze the notebook front end or kernel, forcing termination.

This is a typical *availability* attack pattern in computational notebooks: no file deletion or network exfiltration is required to cause harm.

4 Security-Oriented Rectification Principles

This section lists concrete defensive principles tailored to Wolfram Language.

4.1 P1: Enforce Numeric Evaluation Where Numeric Computation Is Intended

If integrals/recurrences are meant to be numeric:

- Require `NumericQ` inputs.
- Use `N[..., prec]` and avoid leaving free symbols in exponent bases.
- Add guards like `/; infinity ∈ Integers && infinity >= 0`.

4.2 P2: Put Hard Resource Limits Around Untrusted Computation

Wolfram Language provides:

- `TimeConstrained[expr, t, fail]`
- `MemoryConstrained[expr, bytes, fail]`
- `$IterationLimit`, `$RecursionLimit`

For notebook-facing tools, these are essential when loop bounds could be large or user-controlled.

4.3 P3: Fix Scoping: Pass Values, Do Not Rely on Ambient Symbols

Never reference locals from another `Module`. Instead:

- Return intermediate values as an `Association` and pass them forward, or
- compute everything inside one function with properly scoped locals.

4.4 P4: Prevent Expression Explosion

Avoid patterns that create power towers or repeatedly exponentiate growing expressions. If recurrence is necessary:

- Work with `Log` domain when possible,
- clamp magnitudes and abort on overflow,
- use `Chop` or thresholds,
- ensure terms do not become symbolic (see P1).

4.5 P5: Validate Parameters and Use Safe Iteration Constructs

- Replace `For` with `Do` or functional iteration where appropriate.
- Validate bounds: `IntegerQ[n] && 0 <= n <= nMax`.

5 A Safer, Corrected Reference Implementation

Because the original program references many undefined quantities, a "correct" version must first clarify intent. Below is a *rectified structural version* that:

- makes all dependencies explicit as parameters,
- enforces integer bounds,
- enforces numeric evaluation,
- limits time/memory,
- returns intermediate results for auditing.

5.1 Safe Building Block

Listing 2: Safer exponent helper with numeric guard.

```
ClearAll[eSafe];
eSafe[n_?NumericQ, x_Integer?NonNegative] := n^x;
```

5.2 Safe Recurrence With Abort Conditions

Listing 3: Bounded recurrence to mitigate expression/number explosion.

```
ClearAll[boundedRecurrence];
boundedRecurrence[base_?NumericQ, n_Integer?NonNegative,
  maxAbs_: 10^6] :=
 Module[{I = 0., term},
  Do[
    term = (base^i) * (I^i);
    I = I + term;
    If[Not[NumericQ[I]] || Abs[I] > maxAbs, Return[$Failed]],
    {i, 1, n}
  ];
  I
 ];
```

5.3 Rectified "DoubleStruckN" Computation

Listing 4: Rectified and explicit computation of the three integrals.

```
ClearAll[calculateN];
calculateN[
  theta_?NumericQ, lambda_?NumericQ, mu_?NumericQ, nu_?NumericQ,
  xi_?NumericQ, pi_?NumericQ, rho_?NumericQ, sigma_?NumericQ,
  upsilon_?NumericQ, phi_?NumericQ, chi_?NumericQ, psi_?NumericQ,
  n_Integer?NonNegative
] :=
 Module[{base1, base2, base3, r1, r2, r3},
  base1 = theta*lambda*mu*nu;
  base2 = xi*pi*rho*sigma;
  base3 = upsilon*phi*chi*psi;

  r1 = boundedRecurrence[base1, n];
  r2 = boundedRecurrence[base2, n];
  r3 = boundedRecurrence[base3, n];

  If[MemberQ[{r1, r2, r3}, $Failed], Return[$Failed]];

  <|
    "integralRhoGW" -> r1,
    "integralZeta" -> r2,
    "integralOmega" -> r3,
    "doubleStruckN" -> (r1*r2*r3)
  |>
 ];
```

5.4 Safety Wrapper for Execution

Listing 5: Execution wrapper with explicit resource constraints.

```
ClearAll[safeMain];
```

```
safeMain[] :=
 Module[{theta, lambda, mu, nu, n, params, result},
  theta = 1.0;
  mu = 1.0;

  (* IMPORTANT: define all quantities numerically; do not leave symbols free *)
  (* Example placeholders; replace with domain-appropriate values *)
  nu = 0.1;
  lambda = 0.2;

  params = <|
    "xi" -> 0.3, "pi" -> N[Pi], "rho" -> 0.4, "sigma" -> 0.5,
    "upsilon" -> 0.6, "phi" -> 0.7, "chi" -> 0.8, "psi" -> 0.9
  |>;

  n = 8;

  result = TimeConstrained[
    MemoryConstrained[
      calculateN[
        theta, lambda, mu, nu,
        params["xi"], params["pi"], params["rho"], params["sigma"],
        params["upsilon"], params["phi"], params["chi"], params["psi"],
        n
      ],
      200*1024^2, (* 200 MB *)
      $Failed
    ],
    2.0, (* seconds *)
    $Failed
  ];

  Print["Result: ", result];
 ];
```

6 Discussion: What Was "Malicious" vs. What Was "Broken"

The reviewed program is best characterized as:

- **behaviorally dangerous** (availability risk) because it constructs rapidly exploding nested exponent expressions and allows uncontrolled symbolic growth, and
- **technically incorrect** because of scoping errors, undefined variables, duplicated parameters, and ambiguous multiplication.

Even if originally intended as a mathematical exploration, these properties match common denial-of-service patterns in notebook ecosystems.

7 Checklist for Future Wolfram Language Code Reviews

1. Are loop bounds validated and integer?

2. Are all symbols either localized (`Module`) or explicitly passed?
3. Is numeric computation forced when intended (`NumericQ` guards)?
4. Are `TimeConstrained`/`MemoryConstrained` used for untrusted inputs?
5. Is there any recurrence that can create power towers or expression blow-up?
6. Are variables uniquely named (no shadowing/duplication)?

8 Global View: What Can These Programs Do?

Programs of the form shown—loop-driven recurrences that repeatedly apply exponentiation to an evolving accumulator and/or to partially undefined symbolic quantities—often behave as *evaluation bombs* in symbolic computation environments (e.g., Wolfram Mathematica, Maple, SymPy). Even if no explicit file/network primitives are present, they can still be harmful in practice due to their impact on *availability*.

8.1 1. Denial of Service via Resource Exhaustion

A common effect is to force the system to allocate and manipulate extremely large intermediate objects:

- **CPU exhaustion**: repeated exponentiation and expansion has very high computational cost.
- **Memory exhaustion**: symbolic expression trees grow superlinearly (often super-exponentially) in size.
- **UI/kernel freeze**: notebook front-ends become unresponsive as outputs or intermediate expressions become enormous.

This is often sufficient to require killing the computation/kernel, losing session state, or crashing the environment.

8.2 2. Symbolic "Poisoning" and Output Flooding

If the bases of exponentials contain undefined symbols (e.g., `xi`, `rho`, `sigma`, etc.), evaluation tends to remain symbolic. Instead of producing a numeric result, the system constructs a large symbolic object:

- huge printed outputs that flood logs and notebooks,
- expressions too large to inspect or simplify,
- obscured control flow (it becomes hard to see what is being computed).

This can be accidental (bad practice) or intentional (obfuscation).

8.3 3. Unpredictability from Scoping and Name Errors

When code references identifiers that are not in scope (e.g., variables local to another `Module`), the environment typically treats them as global symbols. This makes results depend on ambient session state and on any prior definitions. Such dependence increases fragility and can be exploited to make behavior difficult to reproduce or debug.

8.4 4. If Combined with Side Effects, It Can Become Conventional Malware

The presented pattern is primarily *DoS-like* on its own. However, symbolic systems frequently support:

- file I/O (reading/writing/deleting),
- external command execution,
- network access (HTTP requests, data uploads/downloads),

so a "compute bomb" can be used as a wrapper around more overtly malicious behavior. The core idea is that the victim is distracted by the runaway computation while side-effectful actions occur.

9 Do These Programs Conflate 0 and Infinity?

Not in a rigorous mathematical sense (i.e., they do not prove $0 = \infty$), but they commonly *force the computation into the same boundary regime where* 0*,* ∞*, and indeterminate forms collide.* There are three important points.

9.1 1. Misleading Use of the Word "Infinity"

In many such scripts, a variable is named `infinity` and used as a loop bound, but it is actually assigned a finite numeric value (e.g., 8). This is not infinity; it is merely a truncation parameter. The name encourages the reader to interpret the loop as approximating an infinite process when it is not.

9.2 2. Indeterminate Forms: 0^0 and $0 \cdot \infty$ Appear Naturally

A typical structure is:

$$I \leftarrow I + A^i \, I^i,$$

with initialization $I = 0$ and often with loop indices that include $i = 0$.

The 0^0 problem. If the loop starts at $i = 0$, then I^0 is computed at least once. With $I = 0$ initially:

$$I^0 = 0^0,$$

which is **indeterminate**. Many computer algebra systems return `Indeterminate` for 0^0 (or treat it specially depending on settings). This is a concrete way the program drives the computation into ambiguous boundary semantics.

The $0 \cdot \infty$ family. Even if i starts at 1, later iterations can produce extremely large values (overflowing to ∞ in machine arithmetic), while other factors may remain 0 or become very small. Expressions of the form

$$0 \times \infty$$

are also indeterminate in limit theory. In practice, once a symbolic/numeric system produces `Infinity` (or analogous objects), subsequent multiplications/additions with zero or undefined quantities often yield `Indeterminate` or `ComplexInfinity`.

9.3 3. "Infinity" in Computation vs. Infinity in Analysis

In mathematical analysis, ∞ is not a number; it is a concept used in limits, extended reals, or projective extensions. In computation:

- **Machine overflow** produces sentinel values (e.g., `Infinity`) that do not behave like real numbers.
- **Symbolic infinity** objects have special rewriting rules.
- **Indeterminate** objects propagate through expressions.

These representations can be useful, but careless recurrences can cause them to appear early and contaminate the entire computation.

10 Summary

From a global viewpoint, these programs can:

1. exhaust CPU and RAM by generating power-tower–like growth and massive symbolic expressions,
2. flood outputs and obscure intent via undefined symbols and expression blow-up,
3. behave unpredictably due to scoping/name errors,
4. push the computation into boundary regimes where 0^0, $0 \cdot \infty$, and overflow-related infinities arise.

Thus they do not literally equate 0 and ∞, but they can *repeatedly generate and propagate* the indeterminate forms that arise precisely at the conceptual boundary between 0 and ∞.