

Vuln check

VULN-001 - Alleged Heap Buffer Overflow

AI Claim

The function writes into the output heap buffer **before** verifying enough space is available, causing a **heap buffer overflow**.

Manual Code Analysis

The AI misunderstood the function's **loop memory reservation invariant**.

Loop Safety Invariant

| At the start of every loop iteration, the output buffer has ≥ 260 bytes free.

Actual Control Flow

Step	Operation
1	Reserve space (Line 1225)
2	Perform writes (current iteration)
3	Check remaining capacity (Line 1304)
4	Reserve space for next iteration

AI assumption:

| ✗ "Writes happen first, checks later."

Reality:

| ✓ Reservation always happens BEFORE writes.

Worst-Case Write Analysis

Metric	Value
Reserved buffer space	260 bytes
Maximum possible write	259 bytes
Safety margin	1 byte

✓ **Mathematically impossible** to overflow.

Runtime Validation

Test	Result
AddressSanitizer (1MB, 100 runs)	✓ No overflow
Boundary fuzzing (1–100k)	✓ Passed
Instrumented simulation	<code>min_headroom = 260</code> , <code>max_write = 259</code>

Conclusion

VERDICT: FALSE POSITIVE

No heap overflow exists. and the function uses a **reservation invariant strategy** ensuring capacity before every write.

Classification: AI false positive due to misunderstanding loop invariant pattern.

VULN-002 - Alleged Out-of-Bounds Palette Read

AI Claim: Palette index is not bounds-checked → possible memory disclosure

Final Verdict: ✗ False Positive

AI Reasoning

A code comment says:

| "out of bounds of palette not checked"

AI assumed this means unsafe memory access.

Manual Verification

Property	Value
Palette allocation	1024 bytes
Entries	256 colors × 4 bytes
Initialization	All 256 entries set to black
Max index	255 × 4 + 3 = 1023 (within bounds)

Important: Unused palette entries are **initialized**, not uninitialized memory.

Malicious PNG Test

PLTE chunk: 2 colors only

Pixel indices used: 0, 1, 254, 255

Pixel	Index	Result
(0,0)	0	Valid red
(1,0)	1	Valid green
(0,1)	254	Black (default)
(1,1)	255	Black (default)

No crash

No garbage memory

Manually tested

Risk	Status
Out-of-bounds read	✗ No
Uninitialized memory read	✗ No
Information disclosure	✗ No

Conclusion

VERDICT: FALSE POSITIVE

The comment "out of bounds of palette not checked" is misleading but the code is safe. Manual verification confirms:

- Palette buffer is ALWAYS 1024 bytes (256 entries × 4 bytes)
 - ALL 256 entries are pre-initialized to black (0,0,0,255)
 - Maximum index access: $255 \times 4 + 3 = 1023 < 1024$ (within bounds)

Classification: AI misinterpreted defensive code comment.

VULN-003 - Integer Overflow in Adam7 Calculations

AI Claim: Integer overflow → undersized buffer → heap overflow → RCE

Final Verdict: Bug exists, but NOT exploitable

Root Cause

Type	Size
unsigned	32-bit
size_t	64-bit

Mismatch allows **overflow in intermediate math**.

What Actually Happens

Step	Behavior
Buffer allocation	Uses <code>size_t</code> → correct size
Adam7 offset math	Uses <code>unsigned</code> → can overflow
Overflow result	Produces smaller offsets , not larger

Memory Behavior

```
[===== correctly allocated buffer ======]
==]
```

↑
Writes occur here (WRONG POSITION)
BUT still inside the allocated buffer

Result → **data corruption**, not memory corruption.

Large PNG Test (65535 × 65535 @ 64bpp)

Result	Value
lodepng_decode	Error 83 (allocation failed)
Memory required	8+ GB
Practical exploit	Impossible on typical systems

Security Impact

Risk	Status
Heap overflow	No
Heap metadata corruption	No
Control flow hijack	No
RCE	No
Data corruption	Yes

Final Assessment

Category	Verdict
Bug present	Yes
Security vulnerability	No
Exploitable	No
Severity	Low (correctness bug)

Conclusion:

VERDICT: BUG EXISTS - NOT EXPLOITABLE

The integer overflow in unsigned arithmetic is real but does not lead to the claimed heap overflow or RCE. Key findings:

- Buffer allocation uses `size_t` (64-bit) → correct buffer size
 - Adam7 offsets use `unsigned` (32-bit) → can overflow for huge images
 - Overflow produces SMALLER offsets, not larger
 - Result: Writes occur at wrong positions WITHIN the buffer

VULN-004 - Integer Overflow in `uivector_resize`

Claim

AI reports that `uivector_resize()` multiplies `size * sizeof(unsigned)` without overflow checks, which can lead to undersized heap allocation and subsequent heap buffer overflow during writes.

Manual Code Analysis

The multiplication:

```
allocsize = size * sizeof(unsigned)
```

can overflow when `size` is extremely large. If this occurs, a smaller buffer than intended is allocated, and later writes based on the original `size` would overflow that buffer. Therefore, **the arithmetic flaw and theoretical heap corruption are real.**

Critical Context

Encoder-Only Code

`uivector` exists only when the PNG **encoder** is compiled. PNG **decoding** uses `ucvector`, which does not multiply element size and does not have this overflow pattern. Therefore, a malicious PNG file cannot trigger this code path.

Operation	Uses <code>uivector</code>
PNG Decoding	✗ No
PNG Encoding	✓ Yes

Non-Default Settings Required

The vulnerable branch executes only when compression settings are changed from defaults.

Setting	Required Value	Default
<code>use_lz77</code>	0	1
<code>btype</code>	1	2

With default configuration, this code path is not taken.

Practical Testing

Encoding tests with large images (up to ~858 MB raw data) succeeded without crash on 64-bit systems, confirming that system memory limits are encountered long before overflow conditions.

System	Elements Needed to Overflow	Practical?
64-bit	~4 exabytes	✗ Impossible
32-bit	~1 billion elements	✗ OOM first

On 32-bit systems, memory exhaustion occurs before overflow. On 64-bit systems, required sizes exceed any realistic hardware capability.

What Would Be Required for attack and it's not feasible

1. Application must use lodepng for ENCODING (not just decoding)
2. Attacker must control image data being encoded

3. Image must have >1 billion pixels (on 32-bit) or >4 exabytes (on 64-bit)
4. System must NOT run out of memory first

Final Conclusion

VERDICT: CODE QUALITY ISSUE - NOT PRACTICALLY EXPLOITABLE

The integer overflow bug exists in the encoder resize logic and could theoretically cause heap corruption. However, it is **not reachable via normal PNG decoding**(and `loadpng` mainly decode the png), requires **non-default settings**, and depends on **impossible memory sizes**. As a result, this is a **code quality issue**, not a security vulnerability.

VULN-005 - Decompression Bomb / Resource Exhaustion

Claim

A malicious PNG with extremely high compression ratio can cause excessive memory allocation during decompression because `max_output_size` defaults to **0 (unlimited)**.

Manual Analysis

LodePNG exposes a protection setting:

```
state.decoder.zlibsettings.max_output_size
```

However, its **default value is 0**, meaning **no decompression size limit**. During PNG decoding, compressed image data can expand thousands of times, leading to large heap allocations.

This is not theoretical — it was verified through real decoding tests.

Compression Amplification Observed

PNG Size	Decompressed Output	Ratio
216 B	1 MB	4,630×
666 B	4 MB	6,006×
2.1 KB	16 MB	7,547×
8.2 KB	64 MB	7,726×

Small inputs can expand into tens or hundreds of MB.

API Design Issue

Most developers use the **simple API**, which does **not allow setting limits**.

API	Can Set Limit?	Protected by Default?
<code>lodepng_decode32()</code>	✗ No	✗ No
<code>lodepng_decode24()</code>	✗ No	✗ No
<code>lodepng_decode_file()</code>	✗ No	✗ No
<code>lodepng_decode()</code> + State	✓ Yes	✗ No

The state object is created internally in simple APIs, preventing users from configuring limits.

Official Example Code Test

Using the **official example code**:

Example	Result
<code>lodepng_decode32_file()</code>	Allocated 50 MB
<code>lodepng_decode32()</code>	Allocated 50 MB
<code>lodepng_decode()</code> (advanced example)	Still unlimited

Even recommended examples are vulnerable.

Memory Impact

PNG Size	Output Size	Memory Used
216 B	1 MB	3.8 MB
6.5 KB	50 MB	108 MB
12.9 KB	100 MB	207 MB
25.7 KB	200 MB	390 MB

Demonstrates clear resource exhaustion potential.

Security Impact

Risk	Status
Memory exhaustion	✓ Yes
Triggered by malicious PNG	✓ Yes
Default configuration safe	✗ No
Simple API safe	✗ No
Protection available	⚠ Advanced API only

Final Conclusion

VERDICT: CONFIRMED VULNERABILITY

Confirmed it's decompression bomb vulnerability but:

This is common across PNG libraries (libpng, stb_image, PIL/Pillow all have similar defaults). It represents an industry-wide design pattern where applications are expected to set their own limits and in general cases no limits.

Also other advance api is available in the codebase for decompression but current code is mainly legacy and currently default and in development.

VULN-006 - PLTE Chunk Length Validation

Claim

The code calculates palette size using:

```
palettesize = chunkLength / 3
```

without verifying the length is a multiple of 3. The claim suggests this could cause reading past the end of chunk data, leading to information disclosure or memory corruption.

Manual Analysis

Integer division **truncates toward zero**. For any value:

```
(n / 3) * 3 ≤ n    (always true)
```

This means the code reads **fewer bytes than available**, not more.

If `chunkLength` is not divisible by 3, the remainder bytes are simply ignored.

chunkLength	palettesize	Bytes Read	Result
9	3	9	Exact
10	3	9	1 byte ignored
11	3	9	2 bytes ignored
12	4	12	Exact

No scenario causes reading past `chunkLength`.

Runtime Testing

Malformed PNGs with PLTE sizes not divisible by 3 were decoded under AddressSanitizer.

No crashes or out-of-bounds reads found.

Existing Bounds Validation

```
if(color->palettesize == 0 || color->palettesize > 256) return  
n 38;
```

Case	Outcome
Too small (<3 bytes)	Rejected
Too large (>768 bytes)	Rejected
Non-multiple of 3	Extra bytes ignored

Final Conclusion

VERDICT: FALSE POSITIVE

Ai misunderstands integer division behavior. The code truncates palette size and reads only complete RGB triplets.

And also PNG specification requires PLTE entries to be groups of 3 bytes.

VULN-007 - ICC Profile Size Truncation

Claim

The ICC profile size returned from zlib decompression (`size_t`) is cast to `unsigned`, which could truncate values larger than 4 GB on 64-bit systems, potentially causing memory safety issues.

Manual Analysis

The cast does exist:

```
info->iccp_profile_size = (unsigned)size;
```

If `size > UINT_MAX`, truncation would occur. However, truncation reduces the stored size, meaning subsequent operations process **fewer bytes than allocated**, not more.

This leads to **data truncation**, not memory corruption.

Size Constraints

Item	Value
<code>unsigned</code> max	4 GB
Default <code>max_icc_size</code>	16 MB
Real ICC profile sizes	Usually < 10 MB

Truncation only happens for ICC profiles larger than **4 GB**, which are unrealistic and blocked by default limits.

Default Protection

```
settings->max_icc_size = 16777216; // 16MB default
```

Scenario	Possible?
Truncation under default config	✗ No
Needs ICC profile > 4GB	✗ Not realistic / Most of ICC profile are under < 10mb

Security Impact

Risk	Status
Integer truncation	✓ Yes (theoretical)
Buffer overflow	✗ No
Memory corruption	✗ No
Information disclosure	✗ No
Data truncation	⚠ Possible only with impossible sizes and Largest known ICC profiles: < 10 MB

Final Conclusion

VERDICT: FALSE POSITIVE-

The `size_t` to `unsigned` cast exists but poses no security risk:

Protection Layer 1: Default Limit

max_icc_size = 16777216 (16 MB default)

Any ICC profile >16 MB returns error 113 BEFORE truncation matters

Protection Layer 2: Practical Reality

unsigned max = 4,294,967,295 (~4 GB)

Truncation only matters for profiles >4 GB

Real ICC profiles: typically 1 KB - 10 MB

Largest known ICC profiles: <10 MB

Even IF truncation occurred (impossible in practice), it produces a SMALLER size value, causing under-read not overflow.