LORIKEET SECURITY

Home > Blog > CVE-2026-48689: A One-Byte Heap Overflow i…

← Back to Blog

# CVE-2026-48689: A One-Byte Heap Overflow in FastNetMon's Universal Buffer Class

Lorikeet Security Team May 23, 2026 10 min read

In This Article

LORIKEET SECURITY

| | |
|---|---|
| CVE | CVE-2026-48689 |
| CVSS | **9.8 (Critical)** |
| CWE | CWE-193 (Off-By-One Error), CWE-122 (Heap-based Buffer Overflow) |
| AFFECTED | FastNetMon Community Edition <= 1.2.9 |
| COMPONENT | src/dynamic_binary_buffer.hpp, lines 101, 110, 121, 149, 160 (five separate methods) |
| ATTACK VECTOR | Remote (NetFlow / sFlow / IPFIX / BGP processing) |
| DISCOVERED BY | Lorikeet Security |

This bug is structurally identical to the famous one-byte heap overflows that took out Sendmail in 2003 and dnsmasq in 2017. It is a single character difference in a bounds check, repeated across five methods of the same class, in a buffer abstraction that is used by virtually every protocol path FastNetMon implements.

The class is `dynamic_binary_buffer_t` in `src/dynamic_binary_buffer.hpp`. It is FastNetMon's general-purpose growable byte buffer, used to serialize BGP messages, NetFlow template flowsets, IPFIX records, and Flow Spec NLRI. It has a maximum size (`maximum_internal_storage_size`), a current write offset (`internal_data_shift`), and a heap-allocated backing buffer. Five of its append/copy methods check whether a new write will exceed the maximum, and five of them get the check wrong by exactly one.

LORIKEET SECURITY

compiler hardening, the path from "send a particular sequence of NetFlow templates" to "code execution as the FastNetMon process user" is short.

## The bug

```
// src/dynamic_binary_buffer.hpp around line 110 (append_dynamic_buffer)
bool append_dynamic_buffer(const dynamic_binary_buffer_t& src) {
    size_t length = src.get_used_size();
    if (internal_data_shift + length > maximum_internal_storage_size + 1) {
        return false;                              // <-- BUG: +1
    }
    memcpy(internal_storage + internal_data_shift, src.get_internal_storage(), l
    internal_data_shift += length;
    return true;
}

// Same check, four other methods:
// - append_data_as_pointer    (line ~101)
// - append_data_as_object_ptr (line ~121)
// - memcpy_from_ptr           (line ~149)
// - memcpy_from_object_ptr    (line ~160)

// But append_byte() at line 87 uses the correct check:
bool append_byte(uint8_t b) {
    if (internal_data_shift > maximum_internal_storage_size - 1) {
        return false;                              // CORRECT
    }
    internal_storage[internal_data_shift++] = b;
    return true;
```

`maximum_internal_storage_size + 1`. The difference is that the buggy check permits a write where `internal_data_shift + length == maximum_internal_storage_size + 1` — that is, a write whose last byte lands at index `maximum_internal_storage_size`, which is one past the last valid index of a buffer of size `maximum_internal_storage_size`. One byte past the end.

The `append_byte` method at line 87 uses the correct form. The fact that one method gets the check right and five get it wrong is a strong signal that this is a mechanical bug — probably a copy-paste with an off-by-one ambiguity that the author noticed but did not resolve (see the "TODO: Why +1?" comment).

## The one-byte heap overflow as an exploit primitive

A single-byte overwrite is enough. The classical exploitation technique is to allocate the buffer such that its end abuts a glibc heap chunk metadata structure. On 64-bit glibc, the chunk header for the next allocation is a 16-byte structure: an 8-byte `prev_size` field (used when consolidating with the previous chunk) and an 8-byte `size` field that holds the chunk size plus three flag bits (`PREV_INUSE`, `IS_MMAPPED`, `NON_MAIN_ARENA`). The one byte the attacker can write past the end of `dynamic_binary_buffer_t` lands on the low byte of `prev_size` (if the buffer is exactly aligned) or on the low byte of `size` (if it's offset by 8).

Both targets enable well-documented heap corruption techniques:

- **Off-by-one onto the size field** — expand the apparent size of the next chunk, then trigger a free that consolidates with attacker-controlled memory beyond the chunk boundary. Not

These techniques have been documented in detail since at least 2005 and have produced exploits against real-world targets repeatedly. There is no novel research required to weaponize a one-byte heap overflow in 2026.

## Reachability: everywhere

The buffer class is used pervasively. The reachability surface includes:

| Code path | How an attacker reaches the buggy methods |
|---|---|
| **BGP message encoding (announce/withdraw)** | Any BGP UPDATE message that triggers route processing in FastNetMon causes attribute serialization into a `dynamic_binary_buffer_t` |
| **NetFlow v9 / IPFIX template processing** | Template flowsets are deserialized into and re-serialized from the buffer class. Crafted templates can push the buffer to its maximum. |
| **sFlow sample processing** | sFlow agent data is staged through the buffer class during flow-record extraction. |
| **Flow Spec NLRI construction** | Flow Spec rules are encoded into the buffer when FastNetMon announces them via BGP for mitigation |

LORIKEET SECURITY

## How a fix should look

```
// Replace in all five methods:
if (internal_data_shift + length > maximum_internal_storage_size + 1) { ... }

// With:
if (internal_data_shift + length > maximum_internal_storage_size) { ... }
```

One character. Five places. Done. The TODO comment at line 100 can be removed.

The defensive secondary improvement is to add a unit test that allocates a buffer of a known small size, fills it to exactly the maximum, then attempts one more byte through each of the six append/copy methods. If any of them succeed, the test fails. This is the kind of test that a developer writes once, runs in CI forever, and never has to think about again.

For project maintainers more generally: every fixed-size-with-a-bounds-check class in your codebase should have a unit test that exercises the boundary. The test is one function. It catches every off-by-one in this class structurally. There's no excuse for not having it.

## Compensating controls

- **Restrict every input plugin's reachability.** This bug is reachable from every protocol path; controls that limit any one path help but do not eliminate the surface. Apply the per

checks at every `malloc` / `free` and aborts on detected corruption. The cost is moderate (a few percent of throughput) and the benefit is converting a one-byte overflow from "exploit primitive" to "abort on detection."

- **Build with `-D_FORTIFY_SOURCE=3`** (the strongest fortify level). This adds compiler-inserted bounds checks to `memcpy` calls where the destination size is known at compile time. The `dynamic_binary_buffer_t`'s internal_storage is a member array of known maximum size; FORTIFY can detect over-copies into it.
- **Audit for other off-by-ones.** The "+1" pattern is mechanical. Grep `src/` for `> maximum_internal_storage_size + 1` and any similar boundary expressions. If your tree has been forked or patched locally, check whether the same anti-pattern appears in your local changes.
- **Run AddressSanitizer in test.** ASan trivially catches one-byte heap overflows at the time they happen. A FastNetMon build under ASan in your test environment with a fuzz harness over NetFlow templates will surface this bug within seconds.

## The pattern: copy-paste with off-by-one indecision

This bug has a specific failure mode worth naming: the developer wasn't sure whether the check should be `> max` or `>= max` or `> max + 1`, picked one of the wrong forms, and left a "TODO: Why +1?" comment in the code. The TODO is the smoking gun. The developer knew the check was suspicious. They chose the form that allowed one extra byte rather than the form that disallowed the exact maximum, possibly out of an intuition that "the buffer should be able to hold exactly maximum_internal_storage_size bytes." That intuition is correct, but

central case in any code involving array indexing. If your reviewer can't immediately reason through why the +1 is correct, the check is wrong. Get a second pair of eyes, write a test, do not commit on intuition.

## Disclosure timeline

| Date | Event |
|---|---|
| **2026-04-25** | Vulnerability identified during Lorikeet Security source code audit of FastNetMon Community Edition 1.2.9 |
| **2026-04-25** | CVE ID requested from MITRE |
| **2026-04-25** | Vendor (Pavel Odintsov / FastNetMon LTD) notified at the contact published in `SECURITY.md` |
| **2026-05-22** | CVE-2026-48689 assigned by MITRE |
| **TBD** | Vendor response |
| **TBD** | Fix release |
| **2026-05-23** | Lorikeet Security publishes responsible disclosure report |

LORIKEET SECURITY

28 views

Share

Lorikeet Security Team

PENETRATION TESTING & CYBERSECURITY CONSULTING

LORIKEET SECURITY

*Practical security, written by practitioners.*

## You Might Also Like

Vulnerability Research

**CVE-2026-22769: A Hard-Coded Tomcat Password Gave UNC6201 Root on Dell RecoverPoint for Two Years**

13 min read

Vulnerability Research

**CVE-2026-20127: A One-Byte Bug Cracked the Cisco SD-WAN Control Plane. Here Is Exactly How.**

12 min read

Vulnerability Research

**ESXicape: VM Escape Attacks, VSOCKpuppet, and Why Hypervisor Security Is Under Siege**

16 min read

LORIKEET SECURITY

NEXT →

**CVE-2026-48688: FastNetMon BGP MP_REACH_NLRI IPv6 Decoder Has an Acknowledged TODO**

Penetration Testing

Managed Services

Company

Free Tools

Partners

Industries

Locations

Compliance Testing

LORIKEET SECURITY

Stay Updated

Your email address

Subscribe

Call Us

+1 (689) 202-3940
Kissimmee, FL — HQ

+1 (929) 577-3213
New York, NY

+1 (888) 652-6479
Toll-Free Nationwide

Cookie Settings

Terms of Service

Privacy Policy



We use cookies

We use essential cookies to make our site work. With your consent, we may also use non-essential cookies to improve your experience and analyze traffic. See our Privacy Policy for details.