modzero

TeamSpeak 3

VULNERABILITY DISCLOSURE (REPORT)

TeamSpeak Systems, Inc.

27 May 2026

Changelog

Version	Date	Author	Description
1.0	2026-05-27	Michael Imfeld	Public release

Credits

The research contained in this report was conducted by Michael Imfeld.

Disclosure Timeline

Date	Comment
2026-03-03	modzero contacts TeamSpeak and requests a PGP key and contact mail for encrypted disclosure report delivery.
2026-03-04	TeamSpeak provides public PGP key for secure submission.
2026-03-05	modzero receives key and forwards the encrypted report + proof of concept.
2026-04-08	modzero shares reserved CVE IDs: CVE-2026-4390, CVE-2026-4391, CVE-2026-4392.
2026-04-09	TeamSpeak requests password for proof of concept code; modzero provides it.
2026-04-24	TeamSpeak confirms vulnerabilities and proposes 2–3 weeks for fix development, followed by coordinated disclosure.
2026-04-28	modzero agrees to timeline.
2026-05-21	TeamSpeak proposes a firm timeline for fixes (TS3 Server 3.13.8, SDK 3.5.0), disclosure scheduled for May 27, 2026.
2026-05-22	modzero requests concrete time for alignment; TeamSpeak commits to notify once live.
2026-05-27	Coordinated public disclosure by modzero and TeamSpeak.

Methodology

The findings described in this report are based on dynamic analysis of the *TeamSpeak* 3 server Linux binary (version 3.13.7) and static reverse engineering, supplemented by an unofficial protocol documentation[1] published on GitHub under the organization ReSpeak. All vulnerabilities have been verified through reproducible proof of concept exploits. Implementation details reflect the current understanding derived from observed runtime behavior and may differ from the actual source code implementation.

[1] ReSpeak, TS3 Protocol Paper, *https://web.archive.org/web/20260213132628/https://github.com/ReSpeak/tsdeclarations/blob/master/ts3protocol.md*

1 Summary

TeamSpeak 3 is a widely used VoIP communication platform providing voice and text chat functionality via a dedicated client-server architecture.

As part of this research, the latest stable release of the *TeamSpeak* server software, *TeamSpeak 3* server version 3.13.7 for Linux[2], was analyzed with a particular focus on connection handling and protocol processing mechanisms. The analysis identified three vulnerabilities in the server's connection handshake logic, all of which result in a remotely exploitable denial-of-service condition.

A vulnerability in the handling of duplicate handshake packets can trigger a use-after-free condition due to inconsistent connection management and incomplete resource cleanup. Successful exploitation causes server crashes or leaves the server in a persistent inconsistent state. If the server is password-protected, the attacker needs to know the server password to exploit the vulnerability. Further exploitability beyond denial-of-service and inconsistent states may be possible but has not been confirmed.

A specially crafted ECC key supplied during the connection handshake triggers a heap-based buffer overflow due to a parsing logic flaw. This vulnerability is exploitable by unauthenticated remote attackers and reliably crashes the server process. Further exploitability beyond denial-of-service may be possible but has not been confirmed.

An omitted parameter during the connection handshake causes the server to reach a failing assertion due to insufficient input validation, resulting in a server crash. This vulnerability is likewise exploitable by unauthenticated remote attackers.

As *TeamSpeak 3* servers are typically exposed to the internet, the identified vulnerabilities are directly reachable by remote attackers. Two of the three findings require no authentication and can be triggered with minimal effort, effectively denying service to all connected users.

[2] Obtained from official vendor distribution site (*https://www.teamspeak.com/en/downloads/*), SHA256: 775a5731a9809801e4c8f9066cd9bc562a1b368553139c1249f2a0740d50041e

Findings

Finding	**Use-After-Free via Inconsistent Connection State Management Leading to Denial-of-Service**
CVE-ID	**CVE-2026-4390**
CVSS	🟥 **7.1 High** (CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:L/A:H)

Finding	**Heap-Based Buffer Overflow in ECC Key Parsing Leading to Denial-of-Service**
CVE-ID	**CVE-2026-4391**
CVSS	🟥 **7.5 High** (CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H)

Finding	**Assertion Failure Triggered by Crafted Input Leading to Denial-of-Service**
CVE-ID	**CVE-2026-4392**
CVSS	🟥 **7.5 High** (CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H)

2 Findings

2.1 Inconsistent Connection State Management Leading to Denial-of-Service via Use-After-Free

Class	**CWE-416: Use After Free**
Component	**Packet Retransmission / Acknowledgement Handling**
CVSS	**7.1 High** (CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:L/A:H)

Summary

TeamSpeak 3 server maintains client connections in two separate internal data structures during the connection establishment process. A vulnerability in the handling of duplicate handshake packets causes the server to remove a connection from one data structure while leaving it in the other, creating an inconsistent state. This leads to incomplete connection cleanup where packet retransmission logic frees connection resources that remain accessible through an alternate lookup path. Subsequent packet processing then accesses the freed memory, resulting in a use-after-free condition. Successful exploitation results in server crashes or persistent inconsistent server states, with potential for further exploitation beyond denial-of-service.

Requirements

If the *TeamSpeak 3* server is password-protected, the attacker needs to know the server password.

Details

Based on reverse engineering of the proprietary *TeamSpeak 3* server binary and dynamic analysis during runtime, the server operates in a multi-threaded architecture and processes incoming UDP messages concurrently. The following description reflects the current understanding of the internal design and is derived from observed behavior rather than source-level confirmation.

At a high level, the server maintains state shared across the server instance in a global *server context* object. For each connected client, a *client context* object stores connection-specific state for each connected client. These *client context* objects are stored in two separate lookup structures within the *server context*: one keyed by network endpoint (IP address and port) for clients in the pre-authentication phase, and another keyed by the assigned client ID for clients that have completed the handshake.

During the initial connection phase, a *client context* object is created and inserted into the endpoint-keyed structure. After successful completion of the client initialization step, the client is assigned a unique numerical identifier and the context is additionally registered in the client-ID-keyed structure.

For subsequent incoming packets on that connection, the corresponding *client context* is retrieved from the appropriate lookup structure (endpoint-keyed for pre-handshake packets (client ID = 0) and client-ID-keyed for post-handshake packets) and used to process protocol messages. Reference tracking and lifetime management of both the server context and client context objects appear to be handled across multiple threads, likely using reference-counted or smart-pointer-like mechanisms.

The *TeamSpeak 3* protocol requires command packets sent between peers to be explicitly acknowledged. If an acknowledgment is not received within a predefined time window, the packet is scheduled for retransmission. To implement this behavior, outgoing command packets must be tracked until they are either acknowledged or run into a timeout.

This tracking is implemented using a resend queue maintained within the *global server context*. This queue holds resending packet wrapper objects awaiting acknowledgment or retransmission. Additionally, each *client context* contains a structure (referred to as `m_AckWaitIterList`) that stores references to or iterators over entries in the global resend queue.

Figure 1 provides a simplified overview of these relationships. The actual implementation contains additional levels of indirection, for example, the `m_AckWaitIterList` is organized by packet type with multiple slots per type, but these details have been omitted for clarity.

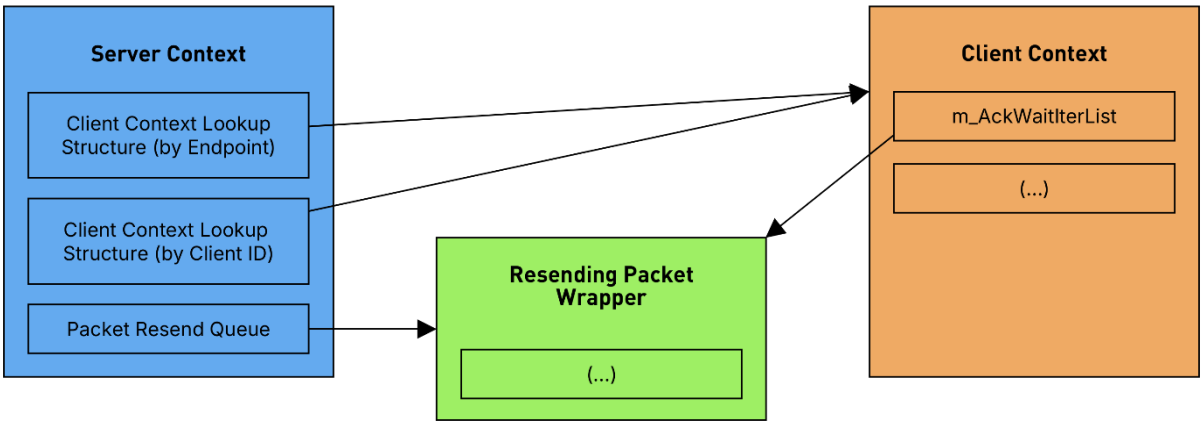

Figure 1 - Simplified overview of data structures involved in packet retransmission.

This design implies shared access to resending packet wrapper objects between the *global server context* and individual *client contexts*. As these structures are accessed and modified by multiple threads, synchronization mechanisms are employed to protect

concurrent access to both the resend queue and the client-specific reference structures.

The responsibility for removing a resending packet wrapper from the respective data structures is mainly shared between two mechanisms. The first is the acknowledgment handler, which removes packets from the queue once an acknowledgment is received from a client. The second is a periodic background task, which also handles retransmissions of pending packets.

The relevant parts of the code (simplified and showing only the essential functionality) are displayed in the following listings. Listing 1 illustrates the logic of the main packet dispatcher responsible for processing acknowledgment packets sent by the client.

```
function handle_incoming_packet(server_context, packet):
    (...) packet validation
    client_context = lookup_client_context()
    if client_context != NULL:
        (...) packet dispatching
        if packet.type == ACK:
            process_received_ack(server_context.resend_queue, client_context, ...)
```

Listing 1 - Main packet dispatcher function.

The handler thread calls `lookup_client_context()` to obtain the *client context*. This function selects the appropriate lookup structure based on the client ID: 0 for pre-handshake connections (endpoint-keyed) or the assigned client ID for post-handshake connections (client-ID-keyed). The handler then proceeds to packet dispatchment and invokes the function responsible for processing acknowledgment packets, depending on the packet type.

```
function process_received_ack(resend_queue, client_context, ...):
    lock(resend_queue.mutex)
    removed_packet = null

    // remove resending packet from m_AckWaitIterList (locks client_context)
    find_and_remove_acked_resending_packet(client_context, ..., removed_packet)

    (...) remove removed_packet from resend_queue

    // free removed_packet and its child objects
    delete removed_packet

    unlock(resend_queue.mutex)
```

Listing 2 - Ack packet handler.

The acknowledgment packet handler acquires a lock on the resend queue and, while holding a lock on the client context, removes the corresponding resending packet wrapper reference from the client's `m_AckWaitIterList`. It then removes the acknowledged resending packet wrapper from the resend queue maintained in the

server context.

Recall from above that in addition to the acknowledgment handler, a periodic background task is likewise responsible for removing resending packet wrappers from the respective data structures. This task iterates through the resend queue to identify packets due for retransmission. For each packet, the task looks up the associated client context using the same client ID-based routing logic: client ID 0 triggers an endpoint-keyed lookup, while non-zero client IDs use the client-ID-keyed structure. If the lookup returns NULL, the task assumes the client has disconnected and removes the resending packet wrapper object from the queue, freeing it along with its associated child objects.

```
function process_resend_queue(resend_queue, ...):
    lock(resend_queue.mutex)

    // traverses resend queue for packets due for retransmission
    for packet in resend_queue:
        if packet.resend_due:
            // lookup client context for packet
            client_context = lookup_client_context()
            if client_context != NULL:
                // return packet for resending
            else:
                // client disconnected
                (...) remove packet from resend_queue
                delete packet

    unlock(resend_queue.mutex)
```

Listing 3 - Periodic processing of resend queue.

The vulnerability arises from an inconsistency between the two *client context* lookup structures when a duplicate *clientinit* packet is processed. When a client successfully completes the high-level handshake by sending a *clientinit* packet, the server assigns a client ID, registers the client context in the client-ID-keyed structure, and sends an *initserver* response.

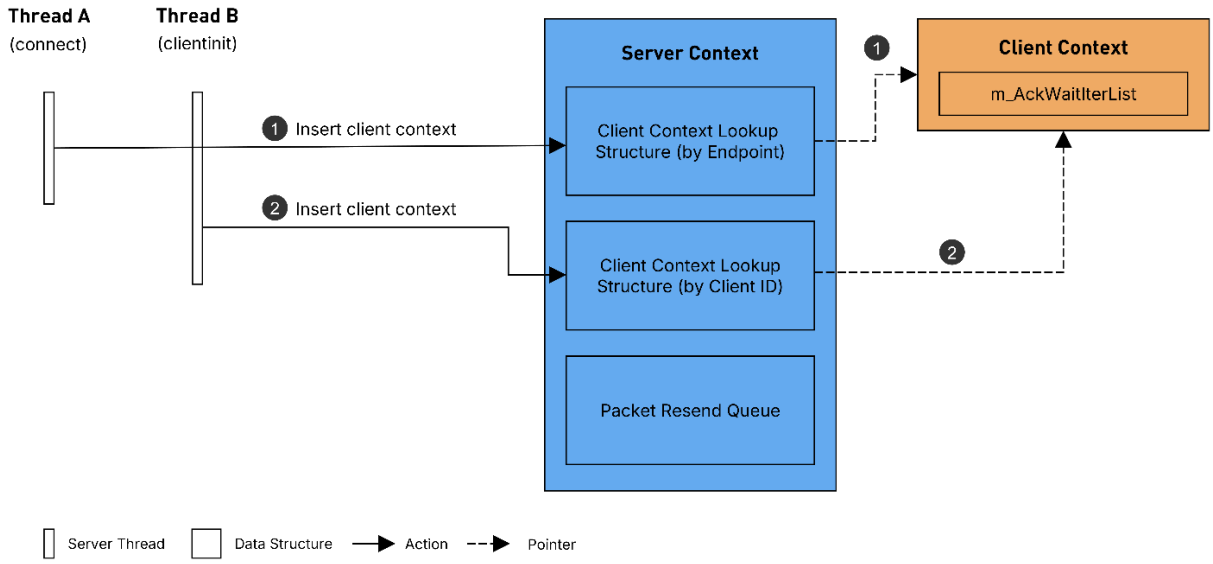

Figure 2 - Initial connection and handshake completion.

If a duplicate *clientinit* packet arrives at the same time or shortly after, the server detects that the handshake counter has already advanced and responds with the error message “please do not hack me” (error code 0×701). The error handling logic constructs an error response packet with the client ID set to 0 (indicating endpoint-keyed routing) and adds it to the resend queue. Critically, the error handler then removes the *client context* only from the endpoint-keyed structure, leaving it in the client-ID-keyed structure. This results in an inconsistent state: the client context is no longer accessible via endpoint lookup but remains reachable through client ID lookup.

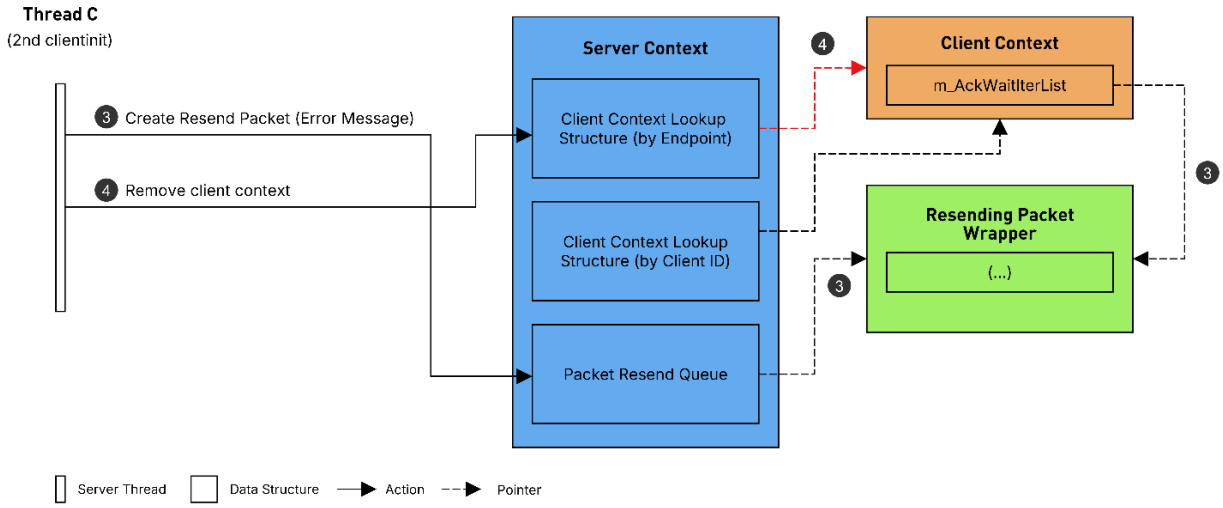

Figure 3 - Second `clientinit` arrives, triggering the error handler.

The queued error response packet remains in the retransmission queue awaiting acknowledgment. If the client does not acknowledge it within the resend timeout, the periodic `process_resend_queue` task attempts to retransmit the packet. The task extracts the client ID from the packet (0) and calls `lookup_client_context()`, triggering an endpoint-keyed lookup. Since the *client context* was already removed from the endpoint-keyed structure, the lookup returns NULL. The resend queue processing task

interprets this as a disconnected client and frees the resending packet wrapper object and its child objects. Because the lookup returned NULL, the task cannot access or clear the corresponding entry in the client's `m_AckWaitIterList`, leaving a dangling pointer behind.

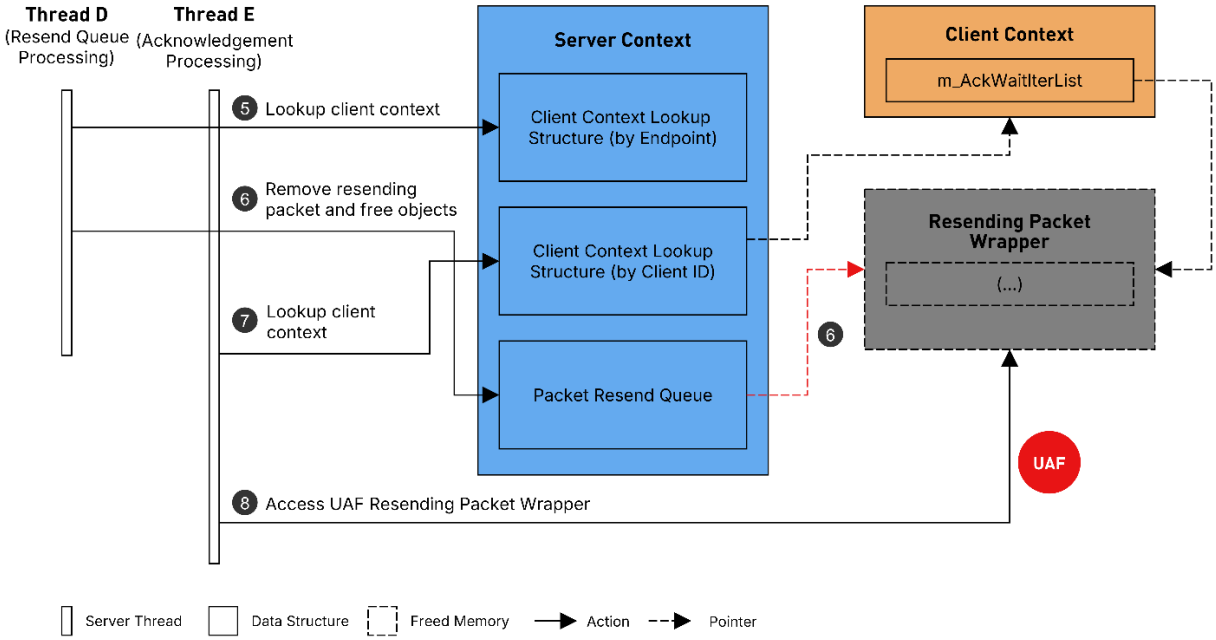

Figure 4 - Resend queue processing frees the resending packet wrapper leading to UAF through m_AckWaitIterList on acknowledgement.

Meanwhile, if an acknowledgment for the error packet arrives after the resend queue task has already freed the resending packet wrapper, the acknowledgment processing thread successfully locates the client context via the client-ID-keyed structure, using the assigned non-zero client ID. The handler then attempts to remove the corresponding resending packet wrapper reference from the client's `m_AckWaitIterList`. At this point, it accesses the already freed packet wrapper object, resulting in use-after-free.

Triggering this vulnerability causes a server crash or leaves the server in a persistent inconsistent state. If the client omits the acknowledgement packet triggering the use-after-free condition, the affected connection remains indefinitely and is not terminated without manual intervention.

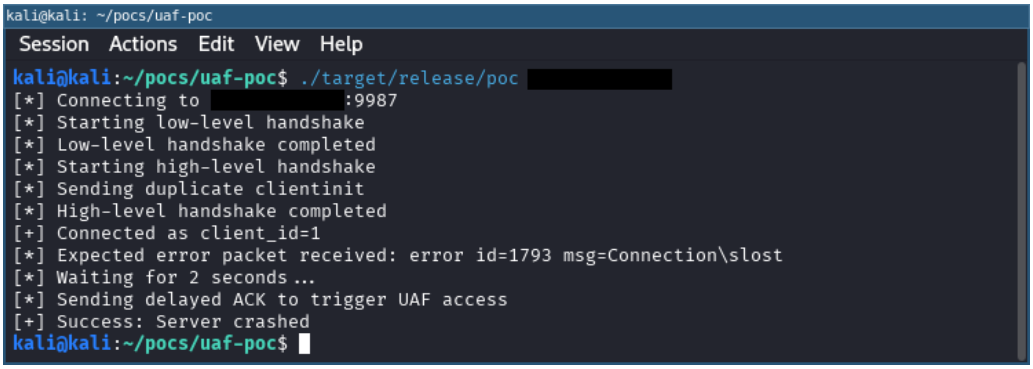

Figure 5 – Proof of concept exploit for use-after-free vulnerability.

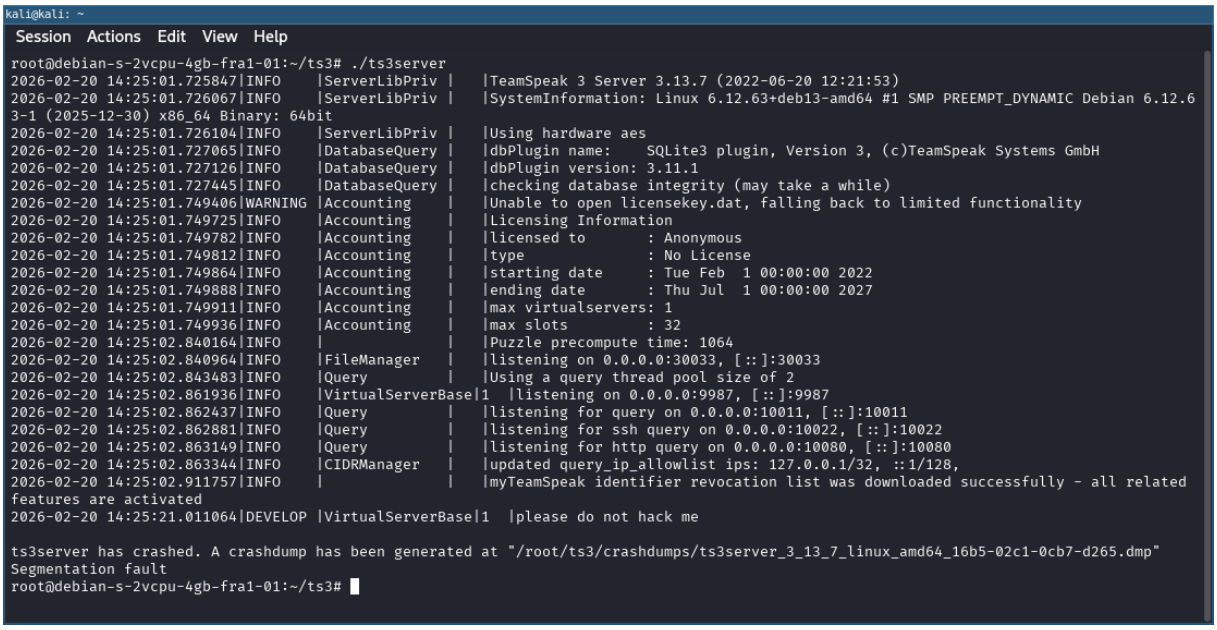

Figure 6 - TeamSpeak 3 server crash caused by use-after-free vulnerability.

Although the currently observed impact is limited to denial-of-service through crashes or connection state corruption, the underlying use-after-free condition constitutes a memory corruption vulnerability that may enable more severe exploitation, including code execution. At the time of writing, no reliable exploitation beyond denial-of-service has been demonstrated.

2.2 Heap-Based Buffer Overflow in ECC Key Parsing Leading to Denial-of-Service

Class	**CWE-122: Heap-based Buffer Overflow**
Component	**Connection Handshake**
CVSS	**7.5 High** (CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H)

Summary

TeamSpeak 3 uses a multi-stage connection handshake between client and server. When a specially crafted elliptic curve public key is supplied during the *clientinitiv* handshake step, a parsing logic flaw in the subsequent *clientek* step triggers a heap-based buffer overflow. This vulnerability allows an unauthenticated remote attacker to cause a denial-of-service condition.

Details

TeamSpeak 3 establishes client connections through a multi-stage handshake in which cryptographic parameters are exchanged between client and server. During the *clientinitiv* step, the client provides its identity public key as a Base64-encoded ASN.1-DER structure in the *omega* parameter. This key represents an elliptic curve cryptography (ECC) public key and is used as part of the ECDH key exchange. In the subsequent *clientek* step, this previously transmitted public key is parsed and processed as part of the cryptographic handshake to derive the shared session secrets.

During testing, a heap-based buffer overflow was identified when a specially crafted ECC public key is provided during the handshake. The key contains a crafted ASN.1 length field that causes the parser to derive an excessively large byte count, which is then passed to a big num import function (presumably *fp_read_unsigned_bin* in TomsFastMath[3]). This function unconditionally increments its internal size counter on each loop iteration without bounds checking against the fixed-size buffer, resulting in an out-of-bounds heap write. The observed behavior is consistent with a known issue in this library, and a corresponding GitHub issue[4] and pull request[5] addressing the problem exist upstream. The identified flaw reliably crashes the server process. Given the nature of the overflow (attacker-controlled length, heap corruption), further exploitation beyond denial-of-service may be possible.

[3] TomsFastMath, Large integer arithmetic library, *https://github.com/libtom/tomsfastmath/*

[4] TomsFastMath, Buffer overflow issue, *https://github.com/libtom/tomsfastmath/issues/14*

[5] TomsFastMath, Upstream fix, *https://github.com/libtom/tomsfastmath/pull/15*

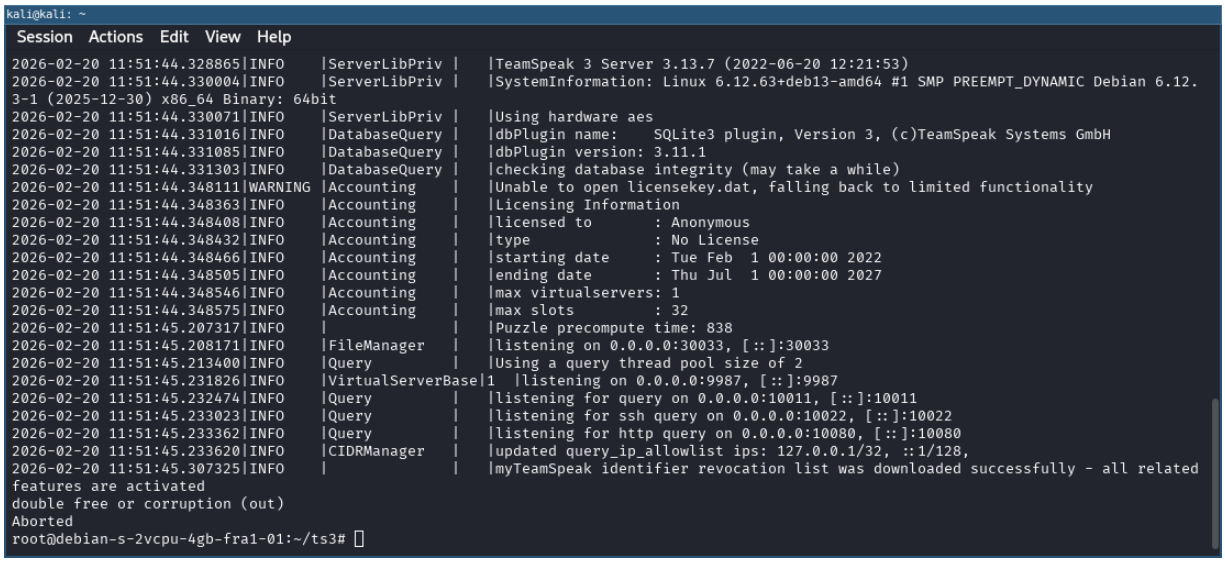

Figure 7 – *TeamSpeak 3* server crash caused by heap-based buffer overflow.

The vulnerable code path is reached prior to password verification. As a result, a remote attacker can trigger the crash without a valid server password, causing a denial-of-service condition.

2.3 Assertion Failure Triggered by Crafted Input Leading to Denial-of-Service

Class	**CWE-20: Improper Input Validation**
Component	**Connection Handshake**
CVSS	**7.5 High** (CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H)

Summary

TeamSpeak 3 performs a multi-stage connection handshake between client and server. If a client omits the required *proof* parameter during the *clientek* handshake step, the server does not properly validate the input and triggers a failing assertion, resulting in a server crash. This vulnerability allows an unauthenticated remote attacker to cause a denial-of-service condition.

Details

TeamSpeak 3 establishes client connections using a multi-stage handshake consisting of several sequential steps, where specific parameters are expected at each stage. According to an unofficial protocol documentation[6] by ReSpeak, the client submits its ephemeral key information to the server during the *clientek* stage using the following command:

```
clientek ek={ek} proof={proof}
```

The ek parameter contains the client's ephemeral public key, encoded as Base64. This key is a temporary key pair generated by the client earlier in the handshake process and is used exclusively for the current session.

The proof parameter is a Base64-encoded signature over the concatenation of the ephemeral public key (ek) and the beta value previously provided by the server during the *initivexpand2* step. This signature is created using the private key of the client's long-term identity key pair and serves to authenticate the ephemeral key and bind it to the client's identity.

It was observed that if a client omits the required *proof* parameter in the *clientek* handshake step, the server does not perform sufficient input validation before using the value internally.

[6] ReSpeak, TS3 Protocol Paper clientek, *https://web.archive.org/web/20260213132628/https://github.com/ReSpeak/tsdeclarations/blob/master/ts3protocol.md#3225-clientek-client---server*

As shown in Figure 8, processing such a malformed handshake leads to a failed assertion, which in turn raises a SIGABRT signal and immediately terminates the server process. The crash occurs reliably upon receipt of the crafted handshake sequence.

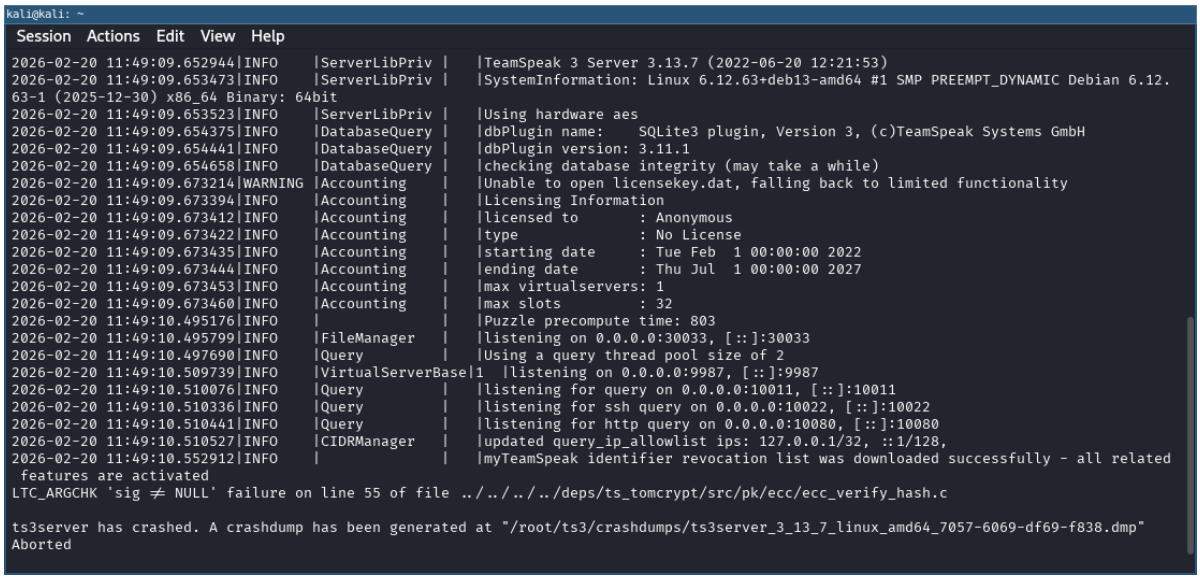

Figure 8 – TeamSpeak 3 server log displaying the failed assertion and crash.

The vulnerable handshake stage is executed prior to authentication and password verification. As a result, the issue can be triggered remotely and affects servers regardless of whether a server password is configured. A remote attacker can exploit this behavior to reliably crash the server, resulting in a denial-of-service condition.