```
0:005> r
rax=000001fda4c0b025 rbx=0000000000000000 rcx=fffffffffff8bc7
rdx=000001fda4c00fc0 rsi=00007ffc19b2632b rdi=00000001007ff440
rip=00007ffc199273ca rsp=00000001007ff3f0 rbp=0000000000000000
 r8=000001fdae625f80  r9=0000000000000001 r10=0000000000000000
r11=00000001007ff110 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei ng nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010286
freerdp2!update_recv_order+0xca:
00007ffc`199273ca 0fb600          movzx   eax,byte ptr [rax] ds:000001fd`a4c0b025=??
0:005> dd rax
000001fd`a4c0b025  ???????? ???????? ???????? ????????
000001fd`a4c0b035  ???????? ???????? ???????? ????????
000001fd`a4c0b045  ???????? ???????? ???????? ????????
000001fd`a4c0b055  ???????? ???????? ???????? ????????
000001fd`a4c0b065  ???????? ???????? ???????? ????????
000001fd`a4c0b075  ???????? ???????? ???????? ????????
000001fd`a4c0b085  ???????? ???????? ???????? ????????
000001fd`a4c0b095  ???????? ???????? ???????? ????????
0:005>
```

This is an out-of-bounds access, positioned to the source location, and the exception is triggered in the update_recv_order function, which handles the order segment data in the rdp message.

```
BOOL update_recv_order(rdpUpdate* update, wStream* s)
{
    BOOL rc;
    BYTE controlFlags;

    if (Stream_GetRemainingLength(s) < 1)
    {
        WLog_Print(update->log, WLOG_ERROR, "Stream_GetRemainingLength(s) < 1");
        return FALSE;
    }

    Stream_Read_UINT8(s, controlFlags); /* controlFlags (1 byte) */

    if (!(controlFlags & ORDER_STANDARD))
        rc = update_recv_altsec_order(update, s, controlFlags);
    else if (controlFlags & ORDER_SECONDARY)
        rc = update_recv_secondary_order(update, s, controlFlags);
    else
        rc = update_recv_primary_order(update, s, controlFlags);

    if (!rc)
        WLog_Print(update->log, WLOG_ERROR, "order flags %02"PRIx8" failed", controlFlags);

    return rc;
}
```

View the code for Stream_Read_UINT8:

```
#define Stream_Read_UINT8(_s, _v) _stream_read_n8(UINT8, _s, _v, TRUE)
```

```
#define _stream_read_n8(_t, _s, _v, _p) do { \
        _v = \
             (_t)(*_s->pointer); \
        if (_p) Stream_Seek(_s, sizeof(_t)); } while (0)
```

S is a wStream type structure pointer:

```
struct _wStream
{
    BYTE* buffer;
    BYTE* pointer;
    size_t length;
    size_t capacity;

    DWORD count;
    wStreamPool* pool;
    BOOL isAllocatedStream;
    BOOL isOwner;
}
```

There are three key fields, buffer, pointer, and length.
Buffer indicates a buffer pointing to the stored message
Pointer points to the location of the current read
The length of the message that The Lengthbuffer points to
The Stream_Read_UINT8 function reads 1 byte through the pointer pointer. So we can know that it should be s->pointer out of bounds that causes Stream_Read_UINT8 cross-border access

But at the beginning of the update_recv_order function, Stream_GetRemainingLength were used to ensure that subsequent reads do not cross the line, why is there an out-of-bounds access?
Check out the code for Stream_GetRemainingLength::

```
static INLINE size_t Stream_GetRemainingLength(wStream* _s)
{
    return (_s->length - (_s->pointer - _s->buffer));
}
```

Here is an integer overflow. The return value of the function is size_t is an unsigned integer, and when the value of (s-pointer - s-buffer) is greater than s-length, a negative value is returned, converted to an unsigned integer, and becomes a larger value, resulting in the explicit pointer having crossed the line, but Stream_GetRemainingLength can't check out the

situation.

So how did pointer cross the line?
View the parent function of update_recv_order update_recv_orders:

```
static BOOL update_recv_orders(rdpUpdate* update, wStream* s)
{
    UINT16 numberOrders;

    if (Stream_GetRemainingLength(s) < 6)
    {
        WLog_ERR(TAG, "Stream_GetRemainingLength(s) < 6");
        return FALSE;
    }

    Stream_Seek_UINT16(s); /* pad2OctetsA (2 bytes) */
    Stream_Read_UINT16(s, numberOrders); /* numberOrders (2 bytes) */
    Stream_Seek_UINT16(s); /* pad2OctetsB (2 bytes) */

    while (numberOrders > 0)
    {
        if (!update_recv_order(update, s))
        {
            WLog_ERR(TAG, "update_recv_order() failed");
            return FALSE;
        }

        numberOrders--;
    }

    return TRUE;
}
```

As you can see update_recv_order is called repeatedly to handle multiple order segments. Could it be that some order segment data caused subsequent pointers to cross the line?

See the other functions called in the update_recv_order function:

```
if (!(controlFlags & ORDER_STANDARD))
    rc = update_recv_altsec_order(update, s, controlFlags);
else if (controlFlags & ORDER_SECONDARY)
    rc = update_recv_secondary_order(update, s, controlFlags);
else
    rc = update_recv_primary_order(update, s, controlFlags)
```

View update_recv_secondary_order:

```
if (Stream_GetRemainingLength(s) < 5) { ... }

Stream_Read_UINT16(s, orderLength); /* orderLength (2 bytes) */
Stream_Read_UINT16(s, extraFlags); /* extraFlags (2 bytes) */
Stream_Read_UINT8(s, orderType); /* orderType (1 byte) */
next = Stream_Pointer(s) + ((INT16) orderLength) + 7;
name = secondary_order_string(orderType);
WLog_Print(update->log, WLOG_DEBUG,  "Secondary Drawing Order %s", name);

if (!check_secondary_order_supported(update->log, settings, orderType, name))
    return FALSE;

switch (orderType) { ... }
```

```
if (!rc)
{
    WLog_Print(update->log, WLOG_ERROR, "SECONDARY ORDER %s failed", name);
}

Stream_SetPointer(s, next);
return rc;
```

Here is a key variable, next. You can see that 2 bytes have been read from the buffer as orderLength, then added to the current position s->pointer, plus 7, and then set next to the new pointer through Stream_SetPointer, pointing to the current position.
When orderLength is oversized, it is bound to result in a situation where (s-pointer-s-buffer) is larger than s-length. Subsequent integer overflows in the Stream_GetRemainingLength lead to a failure of the inspection, which in turn leads to subsequent cross-border access.