CogniMed

Gesellschaft für medizinische
und industrielle Meß- und
Automatisierungstechnik mbH

Peer-Reviews

Fehler finden, die kein Compiler finden kann

1 Einleitung

Moderne C-Compiler wie z.B. der GCC erzeugen nicht nur ausführbaren Code, sondern können auch viele übliche Fehler erkennen und passende Warnungen ausgeben. Mit entsprechenden Parametern aufgerufen wird der Compiler zum Pedanten. Bei einigen Fehlern muss der Compiler aber kapitulieren. Peer-Reviews können diese Fehler finden.

2 Ein Beispiel aus der Praxis

Ein Mikrocontroller-basiertes Gerät sollte in einem Projekt mit einer Schnittstelle für ein externes Speichermedium ausgerüstet werden. Die Schnittstelle würde zunächst nicht genutzt werden, aber die Treiberebene sollte bereits implementiert und minimal getestet werden. Ein Student sollte diese Aufgabe übernehmen.

Die folgende C99-Funktion soll Nutzdaten in mehrere, jeweils vier Byte große Speicherblöcke eines externen Speichermediums schreiben und im Fehlerfall `false`, sonst `true` zurückliefern. Dazu müssen den Nutzdaten vier Bytes mit Kommando, Flags, Nummer des ersten Blocks, und Anzahl der Blöcke vorangestellt werden. Die aufgerufene Funktion `Send()` kümmert sich um die Kommunikation mit dem Speichermedium.

```
#define BLOCK_SIZE 4

  // ...

  bool WriteMultipleBlocks(uint8_t firstBlock, uint8_t blockCount, uint8_t * data)
  {
    if (sizeof(data) > (blockCount * BLOCK_SIZE))
    {
      return false;
    }

    uint8_t buf[(sizeof(data) + 4)];

    buf[0] = CMD_WRITE_BLOCKS;
    buf[1] = CMD_WRITE_FLAGS;
    buf[2] = firstBlock;
    buf[3] = blockCount;

    for (uint8_t i = 4; i < sizeof(buf); i++)
    {
      buf[i] = data[i - 4];
    }

    return Send(buf, sizeof(buf));
  }
```

Die Intention des Autors ist klar: Wenn mehr Nutzdaten geschrieben werden sollen, als in die Blöcke hineinpassen, soll sofort abgebrochen werden. Ansonsten soll ein ausreichend großer Puffer angelegt werden, in den zuerst die Kommando-Bytes, dann die Nutzdaten kopiert werden. Schließlich wird der Puffer an die Kommunikationsroutine übergeben, die ihrerseits bei Erfolg `true` und bei Fehlern `false` liefert.

Gemeinerweise funktioniert diese Funktion mit dem Testcode anstandslos, der einfach nur ein Byte in einen Block schreibt und anschließend wieder ausliest. Der Compiler erzeugte weder Fehlermeldungen noch Warnungen.

3 Der Fehler

Der `sizeof`-Operator liefert die Größe seines Arguments, in Bytes. Hier wird aber schlicht `data` übergeben, ein Zeiger auf die Nutzdaten. Damit liefert `sizeof` die Größe des Zeigers, nicht die Größe dessen, worauf der Zeiger zeigt. Je nach Zielsystem ist so ein Zeiger typischerweise zwei, vier oder acht Bytes groß; unser Zielsystem nutzt einen ARM-Core mit vier Byte großen Zeigern. Egal wie groß die Nutzdaten tatsächlich sind, liefert `sizeof(data)` so immer 4. Das hat Folgen:

- Die `if`-Abfrage ist weitgehend wirkungslos, `sizeof(data)` ist immer 4, `BLOCK_SIZE` ist zufällig ebenfalls 4, so lange `blockCount` nicht Null ist, wird `return false` nicht ausgeführt.
- Der Puffer `buf` hat Platz für `sizeof(data)` plus die vier vorangestellten Bytes, was in jedem Fall acht Bytes sind. Im Puffer ist also Platz für exakt einen Block Nutzdaten. Ein Puffer-Überlauf wird nur zufällig dadurch verhindert, dass die `for`-Schleife nur bis zur Größe des Puffers Nutzdaten in den Puffer kopiert.
- Wenn mehr als ein Block geschrieben werden soll, wird nur der erste Block gesendet. Das ist eine direkte Konsequenz der Begrenzung der `for`-Schleife. Wie verhält sich wohl das externe Speichermedium, wenn weniger Blöcke gesendet werden als in den Kommando-Bytes angekündigt? Das hängt der vom Kommunikationsprotokoll ab. Im schlimmsten Fall interpretiert es folgende Kommandos teilweise als Daten, und Daten der folgenden Kommandos als neue Kommandos. Das könnte zu vollständigem Datenverlust führen.
- Wenn die Nutzdaten kürzer sind als ein Block (vier Bytes), liest die `for`-Schleife über das Ende der Daten hinaus. Da auf dem Zielsystem kein nennenswerter Speicherschutz

vorhanden ist, wird der Puffer schlicht mit bis zu vier Bytes Datenmüll aufgefüllt und auf das Speichermedium geschrieben.

4 Folgefehler

C kann keine Arrays mit erst zur Laufzeit bekannter Größe anlegen, jeder Versuch erzeugt einen Fehler zur Compile-Zeit[1]. Der Puffer `buf` hat, wie oben beschrieben, eine feste und meistens viel zu kleine Größe. Die Intention des Autors war ein Array mit variabler Größe, und die falsche Verwendung von `sizeof` hat verdeckt, dass `buf` eigentlich eine variable Größe haben sollte.

5 Kosten

Die Module mit diesem Fehler galten nach dem zufällig funktionierenden Test als fertig für den nächsten Schritt und bereit für die Verwendung auch in anderen Projekten. Ein formelles Peer-Review wurde nicht verlangt und fand auch nicht statt.

Als die Module in ein anderes Projekt eingebunden werden sollten, fielen diese Fehler zufällig beim Querlesen des Codes auf, und eine kurze Analyse des Codes deckte noch weitere, ähnliche Fehler auf. Das Beheben der Fehler im neuen Projekt kostete ungeplant einen vollen Arbeitstag, dazu kommt noch das Rückübertragen in das alte Projekt. Ein Peer-Review hätte diese Fehler wesentlich früher aufgedeckt und die Module wären nach der Fehlerbehebung wirklich bereit für die Verwendung in anderen Projekten gewesen.

Ein identischer Fehler (`sizeof(pointer)` statt Datengröße) bei einer Änderung in einem vollkommen anderen Projekt erzeugte einen Pufferüberlauf. Allein die Fehlersuche am Gerät kostete einen vollen Arbeitstag, weil der Pufferüberlauf für eine Vielzahl von sehr unterschiedlichen Folgefehlern sorgte. Auch hier hätte ein Peer-Review der Änderung den Fehler deutlich früher aufgedeckt.

6 Reparaturansätze

Wie repariert man nun diese Fehler?

6.1 Ein Sternchen mehr

Wenn `sizeof` immer die Größe seines Arguments liefert, könnte man doch `sizeof(data)` durch `sizeof(*data)` ersetzen, oder? Denn schließlich ist `*data` das, worauf `data` zeigt, also die Nutzdaten.

Leider nein. Es gibt in C schlicht keinen Weg, aus einem Zeiger auf ein Array die Größe des Arrays zu ermitteln. `data` zeigt im Beispiel auf einen `uint8_t`, also ein einzelnes Byte am Anfang der Nutzdaten. Deshalb liefert `sizeof(*data)` schlicht eins, der Puffer würde auf nur fünf Byte verkleinert, und es würde immer nur ein einziges Byte zum Speichermedium gesendet.

6.2 Nutzdatengröße als weiteren Parameter mitgeben

Nach dem Parameter `uint8_t * data` bekommt die Funktion einen weiteren Parameter `size_t dataSize`, der die Größe der Nutzdaten angibt. Dies ist der Standard-Weg, um Puffer variabler Größe an andere Funktionen zu übergeben. So geschieht das bereits bei der aufgerufenen `Send()`-Funktion. Dieser neue Parameter ersetzt in der Original-Funktion `sizeof(data)`.

Leider akzeptiert der Compiler die neue Puffer-Definition nicht, denn Array-Größen müssen zur Compile-Zeit feststehen und können nicht erst zur Laufzeit bestimmt werden. Lesen Sie weiter.

---

[1] Man kann in C *natürlich* zur Laufzeit dynamisch Speicher belegen und diesen Speicher wie ein Array nutzen, z.B. mit Funktionen der `malloc()`-Gruppe oder mit `alloca()`. Man kann für den Zugriff auf diesen Speicher wahlweise Pointer- oder Array-Notation nutzen, was genug Stoff für einen weiteren Artikel ist.

Aber in `typname variable[anzahl];` muss `anzahl` eine zur Compile-Zeit bekannte Konstante sein

6.3 Dokumentation / Problem des Aufrufers

Die Anzahl der Blöcke impliziert, wie viele Nutzdaten geschrieben werden. Man kann schlicht dokumentieren, dass `data` auf `blockCount * BLOCK_SIZE` Bytes Nutzdaten zu zeigen hat. Sind die Nutzdaten länger, wird abgeschnitten, sind sie kürzer, wird mit Datenmüll aufgefüllt. Damit entfällt der `if`-Block, und wir können die Buffer-Definition auf
`uint8_t buf[blockCount * BLOCK_SIZE + 4];` ändern.

Wie zuvor akzeptiert der Compiler die neue Puffer-Definition nicht, weil die Array-Größe zur Laufzeit festgelegt würde.

6.4 Puffer mit fester Größe

Ohne weitere Details zu beachten, kann man die maximale Größe des Puffers festlegen: `blockCount` ist ein `uint8_t`, d.h. es werden maximal 255 Blöcke à vier Bytes geschrieben werden können. Außerdem werden vier Byte für das Kommando benötigt, so dass der Puffer 1024 Bytes groß sein muss. Die geänderte Puffer-Definition `uint8_t buf[255 * BLOCK_SIZE + 4]` akzeptiert der Compiler problemlos. Entsprechend muss die Abbruchbedingung der `for`-Schleife angepasst werden, ebenso die Anzahl der zu sendenden Bytes beim Aufruf von `SendRecive()`, statt `sizeof(data)` ist jeweils gegen `blockCount * BLOCK_SIZE + 4` zu prüfen.

Ganz offensichtlich belegt dieser Puffer während der Laufzeit der Funktion immer 1 kByte RAM, selbst wenn nur ein Byte Nutzdaten geschrieben werden soll. Je nach Mikrocontroller ist 1 kByte RAM schon ein sehr großer Teil des verfügbaren RAMs und kann zu einem Stack-Überlauf führen.

6.5 Statischer Puffer mit fester Größe

Man könnte schlicht eine statische Variable als Puffer benutzen. Dieser Ansatz belegt auch dann 1 kByte RAM, wenn die fragliche Funktion überhaupt nicht benutzt wird. Die Speicherbelegung steht bereits während des Linkens fest, und kann dann gegen die verfügbare Speichermenge getestet werden.

6.6 Dynamischer Puffer

Man könnte `malloc()` benutzen, um sich einen passend großen Speicherblock als Puffer zu beschaffen: `uint8_t * buf = malloc(blockCount * BLOCK_SIZE + 4);`

Prinzipiell ja, aber auf Mikrocontrollern eher nein. `malloc()` kann fehlschlagen, der von `malloc()` beschaffte Speicher muss rechtzeitig und exakt einmal wieder freigegeben werden, und danach darf niemand mehr über den von `malloc()` zurückgegebenen Speicher zugreifen. Kurz: `malloc()` bringt eine große Menge neuer Probleme und Fehlerklassen mit sich, auf die wir in Mikrocontroller-Projekten gerne verzichten.

6.7 Dynamischer Puffer vom Stack

Der lokale Puffer in der originalen Version der Funktion liegt während die Funktion läuft auf dem Stack und wird mit dem Rücksprung wieder freigegeben. Das wurde vom Compiler schon zur Compile-Zeit festgelegt. Sehr ähnlich dazu funktioniert die Funktion `alloca()`, die den Speicherbereich auf dem Stack allerdings erst zur Laufzeit reserviert. Auch dieser Speicherbereich wird mit dem Rücksprung automatisch wieder freigegeben. So wird nur so viel RAM wie nötig belegt, und auch nur so lange wie nötig.

`alloca()` hat allerdings auch einige Nachteile: Der Stack kann unbemerkt überlaufen, `alloca()` hat keinerlei Fehlerprüfung, und abhängig von Zielsystem und Compiler kann es weitere Einschränkungen geben.

6.8 Tatsächliche Reparatur

Dies ist dieselbe Funktion nach der Reparatur:

```
bool WriteMultipleBlocks(uint8_t firstBlock, uint8_t blockCount, uint8_t * data)
{
  if (data == NULL)
  {
    return FALSE;
  }

  size_t bufSize = BLOCK_SIZE * blockCount;
  bufSize += 4; // for command header

  uint8_t * buf = alloca(bufSize);

  buf[0] = CMD_WRITE_BLOCKS;
  buf[1] = CMD_WRITE_FLAGS;
  buf[2] = firstBlock;
  buf[3] = blockCount;

  memcpy(&(buf[4]), data, BLOCK_SIZE * blockCount);

  return Send(buf, bufSize);
}
```

- Die Größe der Nutzdaten wird implizit in `blockCount` übergeben, sie muss ein Vielfaches der `BLOCK_SIZE` sein. Das ist der Dokumentation dieser Funktion zu entnehmen. Ein eigener Parameter für die Größe der Nutzdaten ist im Kontext des Moduls nicht sinnvoll.
- Der dynamische Puffer wird platzsparend mit `alloca()` angelegt, die Dokumentation enthält einen Hinweis darauf, dass die Nutzdaten auf den Stack kopiert werden.
- Die for-Schleife wurde durch `memcpy()` ersetzt. Erstens macht es die Intention deutlicher, zweitens ist `memcpy()` typischerweise hochoptimiert implementiert und sollte selbst im Worst Case nicht langsamer als die `for`-Schleife sein.
- `data == NULL` wird abgefangen.

Natürlich sind auch hier noch weitere Verbesserungen möglich, das würde aber hier den Rahmen sprengen.

A. Foken
Softwareentwickler und IT-Administrator bei der CogniMed GmbH