



OSTBAYERISCHE  
TECHNISCHE HOCHSCHULE  
REGENSBURG

# **STUDIENARBEIT**

Gruber Daniel

## **Vulnerability eingebetteter Systeme**

21. Juni 2022

Fakultät:	Informatik
Abgabefrist:	20. Juli 2022
Betreuung:	Schmidt Jonas

# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>4</b>
<b>2. Vorstellung wichtiger Rahmenbedingungen</b>	<b>4</b>
<b>3. Schwachstellen</b>	<b>7</b>
3.1. memcmp Timing Attacke für Bruteforcing . . . . .	7
3.1.1. Beschreibung . . . . .	7
3.1.2. Beispiel . . . . .	8
3.1.3. Prävention/Schutzmaßnahmen . . . . .	9
3.2. Format String Vulnerability . . . . .	10
3.2.1. Beschreibung . . . . .	10
3.2.2. Beispiel . . . . .	11
3.2.3. Prävention/Schutzmaßnahmen . . . . .	12
3.3. Buffer Overflow (ROP) . . . . .	12
3.3.1. Beschreibung . . . . .	12
3.3.2. Beispiel . . . . .	12
3.3.3. Prävention/Schutzmaßnahmen . . . . .	12
<b>4. Besprechung der möglichen Skalierbarkeit</b>	<b>13</b>
<b>A. Abbildungsverzeichnis</b>	<b>14</b>
<b>B. Literatur</b>	<b>15</b>

# Abkürzungsverzeichnis

**MPU**      Memory Protection Unit

# 1. Einleitung

Embedded Systems befinden sich bereits in vielen Gegenständen und Geräten unseres täglichen Lebens, auch im Auto. Die Vernetzung im Fahrzeug nimmt deutlich zu, weshalb in Deutschland unter anderem die Automobilhersteller im Bezug auf Sicherheit des Fahrzeugs im Fokus sind. Die Automobilhersteller wie BMW, AUDI und Daimler sind erst seit einigen Jahren im Bereich der Softwareentwicklung tätig, worunter insbesondere die Entwicklung eines autonom fahrenden Fahrzeugs und die Entwicklung bzw. mittlerweile Erweiterung an Funktionalität des Infotainmentsystems zählen. Nicht nur, weil diese Automobilhersteller relativ neu in der Softwareentwicklung im Vergleich zu den Technologieriesen wie Google, Facebook und Co. sind, sondern insbesondere wegen der wenigen Schutzmechanismen in Embedded Systems treten hier bereits längst bekannte Schwachstellen vergleichsmäßig oft auf. Darunter fallen Schwachstellen wie die memcmp Timing Attacke als Beispiel für einen Seitenkanalangriff (Side channel attack), der Buffer Overflow und die Format String Vulnerability. In dieser Studienarbeit werden diese drei genannten Schwachstellen detailliert beschrieben, wobei vorneweg konkret auf die STM32 Architektur eingegangen wird. Zusätzlich werden zu jeder der dargestellten Schwachstellen deren mögliche Präventions- und Schutzmaßnahme vorgestellt. Abschließend wird die Skalierbarkeit eines Angriffes basierend auf der Format String Vulnerability auf das Infotainmentsystem eines Fahrzeuges aufgegriffen und besprochen.

## 2. Vorstellung wichtiger Rahmenbedingungen

Die STM32 Mikrocontroller-Familie werden vom europäischen Halbleiterhersteller STMicroelectronics N.V. produziert, welche als eine der ersten Hersteller die CORTEX M3 Lizenz von der Firma ARM erworben hat. Der STM32 Controller zeichnet sich durch eine 32-Bit ARM Cortex-M0/M3/M4 CPU aus, die speziell für Mikrocontroller neu entwickelt wurde. ARM ist ein Reduced Instruction Set Computer (RISC), welche den Vorteil von insbesondere einen kompakten Befehlssatz sowie vielen Registern hat.

Ein Hauptbestandteil des Cortex M3 Prozessors, wie beim STM32F103C8T6 vorhanden, ist die dreistufige Pipeline, die auf der Harvard Architektur basiert. Hierbei existieren, wie für die Harvard Architektur typisch, verschiedene Busse

für Befehle und Daten, welches ermöglicht zugleich Befehle und Daten zu lesen bzw. Daten in den Speicher zurückzuschreiben. Aus Programmiersicht ist die CPU aber ein Von-Neumann Modell, da zwar die Trennung zwischen Befehls- und Datenbus existiert, jedoch sowohl Befehle und Daten im gleichen Speicher (Flash) liegen und somit der Adressraum dementsprechend linear programmiert werden kann. Hier spricht man oft von einer Adeptive Harvard Architektur, da es zwar verschiedene Busse für Daten und Befehle gibt, jedoch keine strikte Trennung zwischen Daten- und Befehlsadressraum gegeben ist. Zudem ist hier kein getrennter physikalischer Speicher für Daten und Befehle vorhanden, denn beides befindet sich im Flash, worauf sowohl der Datenbus (DBUS) und Befehlsbus (IBUS) zugreift, wie auf nachfolgender Abbildung zu sehen. Dabei

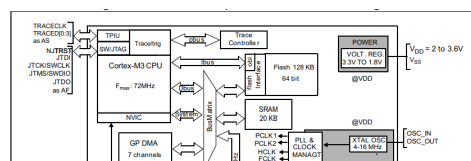


Abbildung 1: STM Architecture

sichert man sich den Vorteil der Harvard-Architektur, dass gleichzeitiges Laden von Befehlen und Daten für bessere Performance möglich ist, jedoch verliert man den Nachteil durch den gemeinsamen Adressbereich bzw. Speicherbereich wie in Neumann, dass der Programmcode manipuliert werden kann. Dies ist insbesondere bei der Schwachstelle Buffer Overflow bzw. Return Oriented Programming von Bedeutung. Des Weiteren besitzen die meisten STM32, insbesondere der in der Übung verwendete STM32F103C8T6, eine Memory Protection Unit (MPU). Diese ermöglicht es, ein eingebettetes System robuster und sicher zu machen, indem beispielsweise der SRAM bzw. Bereiche vom SRAM als nicht-ausführbar definiert werden können.

Das Speichermodell bzw. der Adressierungsbereich von möglichen 4GB der CPU ist in der linken Abbildung dargestellt. Teil dieses Adressierungsbereichs sind der Code, der sich im Flash befindet, und der SRAM, welche beide in der rechten Abbildung dargestellt sind. Dabei ist insbesondere wichtig, dass der Stack nach unten, d.h. von höheren zu niedrigen Adressen wächst, was es ermöglicht, return Adressen und andere Bereiche im SRAM über einen Buffer Overflow oder eine Format String Vulnerability für Angriffe auszunutzen.

Auch die Funktionsweise des LR Register ist für die konkrete Ausnutzung nachfolgender Schwachstellen bedeutend. Denn dieses speichert die Return ad-

Figure 2. Cortex-M0+/M3/M4/M7 processor memory map

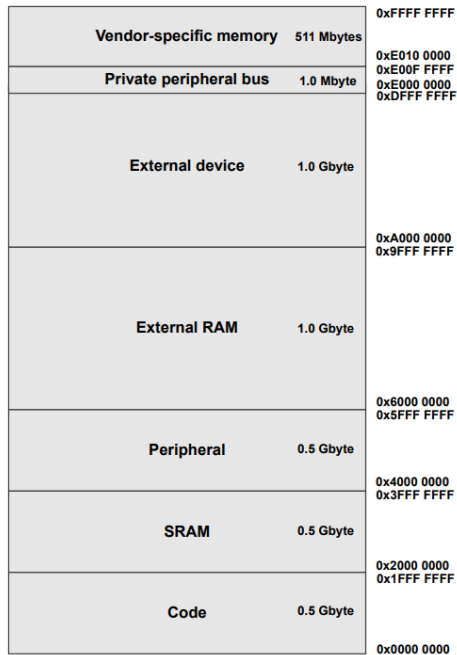


Abbildung 2: Memory map

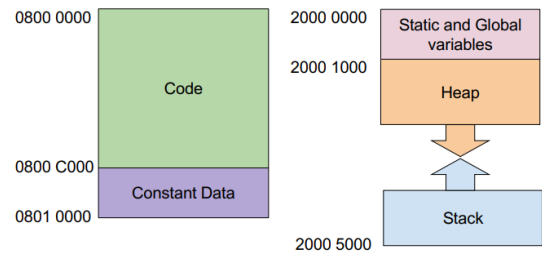


Abbildung 3: Flash und SRAM

Adresse der Funktion. Erst wenn mehrere Funktionen vorhanden sind, wird die return Adresse auf dem Stack gespeichert und muss von diesem wieder geladen werden. Deswegen müssen bei der Überschreibung von Return Adressen zwei Funktionen, wobei eine die andere aufruft, um die Überschreibung der returnadresse der äußeren Funktion zu ermöglichen, vorhanden sein.

Register	AKA	Use
r0		Return value, first function argument
r1-r3		Function arguments and general scratch
r4-r11		Saved registers
r12	ip	Intra-procedure scratch register, rarely used by the linker
r13	sp	Stack pointer, a pointer to the end of the stack. Moved by push and pop.
r14	lr	Link register, storing the address to return to when the function is done. Written by "bl" (branch and link, like function call), often saved with a push/pop sequence, read by "bx lr" (branch to link register) or the pop.
r15	pc	Program counter, the current memory address being executed. It's very unusual, but handy, to have the program counter just be another register—for example, you can do program counter relative addressing very easily, by just loading from [pc+addr].

Abbildung 4: ARM Register Set

Außerdem ist hier zu erwähnen, dass die STM32 Mikrocontroller-Familie auf Little Endian setzt, d.h. das niederwertigste Byte befindet sich an der niedrigsten Adresse. Die Abspeicherung in Little Endian spielt insbesondere für die Schwachstelle Buffer Overflow eine wichtige Rolle, da beim Auslesen des Speichers dies zu berücksichtigen ist.

Wie viele Embedded Systems hat auch der STM32 kein Betriebssystem, das weitere Schutzmechanismen bieten würde.

## 3. Schwachstellen

### 3.1. memcmp Timing Attacke für Bruteforcing

#### 3.1.1. Beschreibung

Die memcmp Timing Attacke ist ein typischer Seitenkanal-Angriff. Diese Art von Angriffen basieren auf Informationen, die von der konkreten Implementierung eines Systems abhängen. Bei der memcpm Timing Attacke basiert dies auf dem Wissen über die Softwareimplementierung eines Vergleichs von Speicherbereichen. Denn im Fall, dass eine Speichervergleichsfunktion so implementiert ist, dass beim ersten nicht übereinstimmende verglichenenen Zeichen von der Funktion 'false' zurückgegeben wird, benötigt der Vergleich unterschiedlich lange, je nach Anzahl richtiger Buchstaben einer Zeichenkette. Hier wird konkret der Aspekt der Zeit ausgenutzt, denn die Dauer der Funktion hängt von den zu vergleichenden Speicherbereichen ab. Je länger die Funktion benötigt, desto mehr Buchstaben waren beim entsprechenden Vergleich richtig. Diese Information der Dauer einer Funktion je nach Vergleich kann man nun ausnutzen, um Bruteforcing bei Passworteingaben deutlich zu optimieren. Bei 'normalen' Bruteforcing müssen alle Kombinationen durchprobiert werden, d.h. bei einem Passwort der Länge 6 müssen bis zu  $|A| * |A| * |A| * |A| * |A| * |A| = |A|^6$  Möglichkeiten durchprobiert werden.  $|A|$  ist die Mächtigkeit der möglichen Eingabezeichen. Dahingegen kann bei einer memcmp Timing Attacke Stelle für Stelle durchprobiert werden, und die Auswahl für die jeweilige Stelle, die am längsten benötigt hat, wird als 'richtig' übernommen, denn dann hat die Vergleichsfunktion für die jeweilige Stelle einen erfolgreichen Vergleich durchgeführt. Dies führt dazu, dass die nächste Stelle überprüft wird, was bedeutet, dass die Funktion dafür mehr Zeit braucht. Insgesamt führt die memcmp Timing Attacke also zu einer erheblichen Verbesserung, indem beim Fall der Passwortlänge von 6 nur bis zu  $|A| + |A| + |A| + |A| + |A| + |A| = 6 * |A|$  Möglichkeiten durchprobiert werden müssen.

#### *Anmerkung*

In der Realität liegt solch ein Vergleich im Bereich von Nanosekunden, da nur wenige Clock Cycles für den Vergleich benötigt werden. Dies bedeutet, dass der Delay über ein USB Kabel deutlich größer ist (im Millesekunden Breich) als die Dauer des Vergleichs. Aus diesem Grund werden für solche memcmp

Timing Attacks Oszilloskope oder Logic Analyzers benötigt, um den Zeitunterschied für den Vergleich am Embedded System zu messen.

### 3.1.2. Beispiel

In diesem Abschnitt wird ein repräsentatives Beispiel für oben genannte Schwachstelle dargestellt. Der Einfachheit halber wird ein PIN Vergleich der begrenzten Länge 4 durchgeführt, wobei das Alphabet 0-9 ist, d.h. eine Mächtigkeit von  $|A| = 10$  besitzt. Zudem wird die Annahme getroffen, dass der Pin Vergleich erst nach vollständiger Pineingabe erfolgt. Dabei wird folgender Code Ausschnitt für die Überprüfung des PINs verwendet.

```
1 bool pin_correct(char *input){
2     char *correct_pin = "1337";
3     for (int i = 0; i < 4; i++){
4         if (input[i] != correct_pin[i]){
5             return false;
6         }
7     }
8     return true;
9 }
```

Für reines Raten, d.h. Bruteforcing ohne weitere Kenntnisse, sind eine gesamte Anzahl von  $10 * 10 * 10 * 10 = 10^4$  Kombinationen möglich. Um die Anzahl der Kombinationen deutlich zu reduzieren kann man den Vorteil des Wissens über die obige Funktion nutzen und damit die memcmp Timing Attacke verwenden. Denn obiger Code Ausschnitt gibt beim ersten nicht korrekten Zeichen 'false' zurück, weshalb die Ausführungsdauer der Funktion von der Anzahl der richtig eingegebenen Pin Stellen abhängt. Dafür geht man Stelle für Stelle durch und überprüft angefangen bei der ersten Stelle für jede mögliche Eingabe von 0-9, welche die längste Zeit benötigt. Denn wenn die Stelle richtig ist, war der Vergleich richtig und die Funktion wird die nächste Stelle überprüfen, was mit einer längeren Dauer für die Funktion einhergeht. Konkret für die erste Stelle werden also alle Möglichkeiten durchgetestet von 0000, 1000, 2000 bis 9000, wobei für jeder dieser Eingaben eine Zeitmessung durchgeführt wird. Folgende zwei Abbildungen stellen für die erste Eingabestelle dar, wie sich die Vergleichszeit im korrekten Fall ( $t_{correct}$ ) zum Fehlerfall ( $t_{bad}$ ) unterscheidet. Da der korrekte Pin 1337 ist, wird für die Eingabe 1000 die Vergleichszeit länger dauern, wie in  $t_{correct}$  dargestellt. Für alle anderen Möglichkeiten wird die



linke Abbildung mit  $t_{bad}$  zutreffen. Dieser Angriff wird für jede Möglichkeit der nächsten Stelle bis zur letzten Stelle durchgeführt ausgehend von der korrekten Eingabe der jeweils vorherigen Stellen. Bei der letzten Stelle ist die Zeitmessung überflüssig, denn im korrekten Fall hat man das System entsperrt. Der Vorteil dieser Methode ist, dass die PIN Stellen sequentiell ausgehend vom Wissen über die Position richtig erraten werden. Damit erreicht man, dass die maximale Anzahl an Kombinationen maximal  $10 + 10 + 10 + 10 = 4 * 10 = 40$  beträgt. Das bedeutet, dass die Möglichkeiten bei der memcmp Timing Attack für Bruteforce gegenüber reinem Bruteforce nur noch  $\frac{40}{1000} = \frac{4}{100} = 4\%$  aller Möglichkeiten betragen.

### 3.1.3. Prävention/Schutzmaßnahmen

Für obige Funktion gibt es eine Vielzahl von Schutzmaßnahmen, die im Wesentlichen solche Angriffe deutlich erschweren, aber nicht 100%ig verhindern. Bei der Annahme, dass das Passwort in Klartext überprüft wird und nicht als gehashter Wert, werden insgesamt 4 Schutzmaßnahmen vorgestellt. Die erste Schutzmaßnahme zielt auf eine korrelationslose bzw. konstante Zeit bei der Überprüfung ab. Dies wird erreicht, indem unabhängig von einer falschen Stelle immer alle Stellen überprüft werden und nachfolgend erst das Ergebnis des Vergleichs zurückgegeben wird. Hierbei wäre für oben dargestellten Code eine wesentliche Änderung nötig, nämlich die Verwendung einer booleschen Variable, die defaultmäßig true ist und bei einem fehlerhaften Vergleich auf false gesetzt wird. Dabei ist zu beachten, dass alle Stellen überprüft werden und erst am Ende das Ergebnis des Vergleichs zurückgegeben wird.

```
1  bool pin_correct(char *input){
2      char *correct_pin = "1337";
3      bool test = true;
4      for (int i = 0; i < 4; i++){
5          if (input[i] != correct_pin[i]){
6              test = false;
7          }
8      }
9      return test;
10 }
```

Eine ähnliche Schutzmaßnahme, die zwar nicht auf konstante Zeit setzt, sondern auf Randomisierung von Zeit, kann durch Hinzufügen von randomisierten

sleeps implementiert werden. Dies erschwert die Korrelation von gemessener Zeit und korrekter bzw. fehlerhafter Eingabe.

## 3.2. Format String Vulnerability

### 3.2.1. Beschreibung

Eine Format String Vulnerability tritt auf, wenn eine Benutzereingabe als Befehl interpretiert wird. Weitergeführt kann ein Angreifer dies ausnutzen, um Code auszuführen, den Stack auszulesen oder gezielt das Programm durch einen Segmentation Fault zum Absturz bringen. Diese Schwachstelle kann man beispielsweise in printf, fprintf und weiteren print Funktionen ausnutzen. Das erste Argument einer printf Funktion ist der sogenannte Format String und die im Format String enthaltenen weiteren Parameter, die mit % beginnen, wie %s, %d, %x und weitere, werden durch nachfolgende Argumente ersetzt. Die nachfolgende Abbildung verdeutlicht dies, wobei *name* ein string und *age* eine integer Variable ist, die in den entsprechenden Paramtern %s und %d als Argumente ersetzt werden.

Das Diagramm zeigt den Code `printf("Hello %s, your age is %d.", name, age);`. Eine Klammer unter dem Format String `"Hello %s, your age is %d."` ist mit *Format String* beschriftet. Ein Pfeil, der von `%d` zum Parameter `age` zeigt, ist mit *weitere Parameter* beschriftet.

Abbildung 5: printf - Format String

Falls die printf - Funktion unsicher programmiert ist, wie in folgenden Code-Ausschnitt zu sehen, wird diese Funktion ohne explizite Parameter aufgerufen, weshalb die Werte von den Registern bzw. weiterführend vom Stack ausgelesen und als Hexadezimal zurückgegeben werden.

```
int main(){
    char *input;
    scanf("Enter any input", input); // Input vom User z.B.: %x%x%x%x
    printf(input);
}
```

Durch das Auslesen von Registern, Stack und Speicher kann ein Angreifer wertvolle Informationen über das laufende Programm gewinnen. Dazu können beispielsweise Passwörter zählen, die im Speicher abgespeichert.

Des Weiteren ist es möglich, wie anfangs erwähnt, eigenen eingegebenen Code auszuführen. Dabei ist oft das Ziel, ein Shell zu öffnen, auf der weitere Aktionen ausgeführt werden können. Hierbei muss man erst den Shell start in Assembly suchen. Mit der Format String Vulnerability muss man darauffolgend mit %x die return Adresse der jeweiligen print Funktion herausfinden. Nachfolgend kann die Schwachstelle ausnutzen, indem man im Input die Adresse der return Adresse von printf schreibt und daraufhin %n ausnutzt um eben an diese Adresse mit der Adresse zum Ausführen der Shell überschreiben.

### 3.2.2. Beispiel

Folgendes Beispiel konzentriert sich auf die beispielhafte Anwendung der Format String Vulnerability in Bezug auf das Auslesen von Speicher. Der folgende Code zeigt ein Beispielprogramm, dass eine printf Implementierung mit oben beschriebener Schwachstelle aufweist.

```
1  int main(){
2      char* correct_password = "f0rm4tS7r!ng";
3      char* username;
4      int age;
5      // User Input
6      scanf("Hello, please first enter your age: ", &age);
7      scanf("Enter your username: ", username);
8      // For taking advantage of this vulnerability: 1. enter a number for which
9      // Output
10     printf("Your age is: %d", age);
11     printf("Welcome ");
12     printf(username); // FORMAT STRING Vulnerability
13 }
```

Hierbei wird der Nutzer nach zwei Eingaben gefragt, nämlich dem Alter und dem gewünschten Benutzernamen. Letztere Eingabe weist die entsprechende Format String Schwachstelle auf. Hierbei kann der Nutzer bzw. der Angreifer dies ausnutzen, indem er als Alter eine beliebige Zahl eingibt, die als Adresse für die auszusende Speicherzelle dient. In der zweiten Eingabe, der Benutzername eingabe, kann man nun mit entsprechend vielen %d und dann einem %s, welches sich genau an der Position des vorher gegebenen Alters eingibt. Beispielsweise kann die Eingabe dann wie folgt aussehen: %d%d%s. Das eingegebene Alter bzw. die Adresse der Speicherstelle, die ausgelesen werden

soll, wird mit %s als Pointer interpretiert und ausgegeben. Hiermit kann man beispielsweise nun den RAM, den Flash oder andere Speicherbereiche auslesen.

### 3.2.3. Prävention/Schutzmaßnahmen

Für die Format String Vulnerability existiert eine einfache sehr effektive Schutzmaßnahme, nämlich das sichere Programmieren, indem man die Parameter %s, %d, ... korrekt benutzt. Weitere Schutzmaßnahmen in der Software könnten zudem noch sein, die Eingabe des Nutzers zu überprüfen, und derartige möglicherweise schädliche Eingaben nicht zuzulassen.

## 3.3. Buffer Overflow (ROP)

### 3.3.1. Beschreibung

Ein Buffer Overflow tritt dann auf, wenn ein Programm mehr Daten in einen Buffer, beispielsweise ein Array in der Programmiersprache C, versucht zu speichern, als dieser umfasst. In einem konkreten Angriff kann man diese Schwachstelle nutzen, um Assembly in den Stack zu schreiben, um mit diesen beispielsweise eine (admin) shell zu öffnen. Dabei wird der Buffer solange überschrieben, bis das Link Register überschrieben wird, und schreibt an dessen Stelle gewünschten Assembly Code.

### 3.3.2. Beispiel

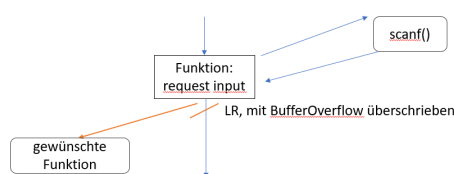


Abbildung 6: Buffer Overflow

### 3.3.3. Prävention/Schutzmaßnahmen

## **4. Besprechung der möglichen Skalierbarkeit**

## A. Abbildungsverzeichnis

1.	STM Architecture . . . . .	5
2.	Memory map . . . . .	6
3.	Flash und SRAM . . . . .	6
4.	ARM Register Set . . . . .	6
5.	printf - Format String . . . . .	10
6.	Buffer Overflow . . . . .	12

## **B. Literatur**

[1] Dr. Nemo: *Submarines through the ages*, Atlantis, 1876.

## **Erklärung**

1. Mir ist bekannt, dass dieses Exemplar des Praktikumsberichts als Prüfungsleistung in das Eigentum der Ostbayerischen Technischen Hochschule Regensburg übergeht.
2. Ich erkläre hiermit, dass ich diese Studienarbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

---

Ort, Datum und Unterschrift

Vorgelegt durch:	Gruber Daniel
Matrikelnummer:	3214109
Bearbeitungszeitraum:	14. März 2022 – 20. Juli 2022
Betreuung:	Schmidt Jonas