OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

STUDIENARBEIT

Gruber Daniel

Vulnerability eingebetteter Systeme

25. Juni 2022

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

Inhaltsverzeichnis

Abkürzungsverzeichnis

MPU	Memory Protection Unit
DEP	Data Execution Prevention
ROP	Return Orientated Programming

1. Einleitung

Embedded Systems befinden sich bereits in vielen Gegenständen und Geräten unseres täglichen Lebens, auch im Auto. Die Vernetzung im Fahrzeug nimmt im aktuellen Jahrzent deutlich zu, weshalb in Deutschland unter anderem die Automobilhersteller im Bezug auf Sicherheit des Fahrzeugs im Fokus stehen. Die Autombilhersteller 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 verlgeichsmäßig oft auf. Darunter fallen Schwachstellen wie die memcmp Timing Attacke als Beispiel für einen Seitenkanalangriff(Side channel attack), Buffer Overflows und Format String Vulnerabilities. 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 Vulnerabilty 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 Programmierersicht 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 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

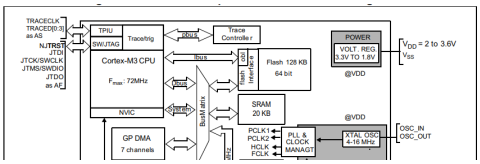

Abbildung 1: STM Architecture

sichert man sich den Vorteil der Harvard-Architektur, dass gleichzetiges 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 Orientated 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 Vulnerabilty für Angriffe auszunutzen.

Auch die Funktionsweise des LR Register in Kombination mit dem PC Counter Register ist für die konkrete Ausnutzung nachfolgender Schwachstellen bedeu-

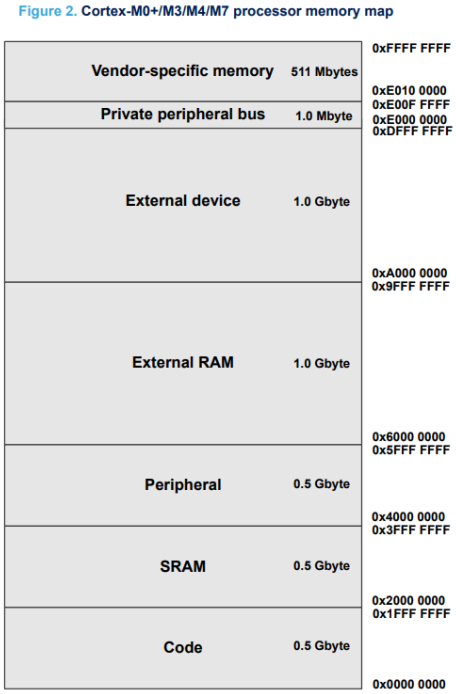

Abbildung 2: Memory map

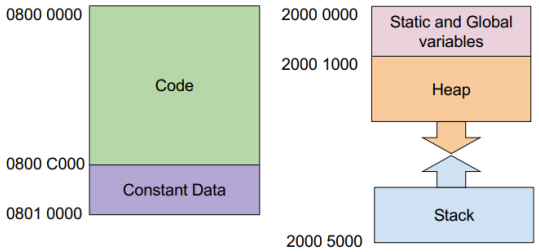

Abbildung 3: Flash und SRAM

tend. Denn das LR speichert die Return addresse der Funktion, welche nachfolgend dann in das Program Counter (PC) geladen wird. Der PC gibt an, wo das Programm fortgesetzt wird. Erst wenn mehrere Funktionen vorhanden sind, wird die return Adresse auf dem Stack gespeichert und muss von diesem wieder in das LR geladen werden, welches dann wiederum in den PC geladen wird. 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.

r0 - r12	General Purpose Register
r13	Stack Pointer (SP)
r14	Link Register (LR)
r15	Program Counter (PC)

Abbildung 4: ARM Register Set

Außerdem ist hier zu erwähnen, dass die STM32 Mikrocontroller-Familie standardmäßig 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 die meisten Embedded Systems hat auch der STM32 kein Betriebssstem, 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 benötigte Zeit eines Vergleichs von Speicherbereichen, welche in der Software implementiert ist. Denn im Fall, dass eine Speichervegleichsfunktion so implementiert ist, dass beim ersten nicht übereinstimmendem verglichenen 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 betrachtet werden, d.h. bei einem Passwort der Länge 6 sind $|A|*|A|*|A|*|A|*|A|*|A| = |A|^6$ Kombinationen möglich, wobei $|A|$ die Mächtigkeit der möglichen Eingabezeichen ist. 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 Vegleichsfunktion 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 $|A| + |A| + |A| + |A| + |A| + |A| = 6 * |A|$ Kombinationen möglich sind.

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 Attacken 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 beschriebene Schwachstelle dargestellt. Der Einfachkeit 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 $10*10*10*10 = 10^4$ 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 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 das obere Diagramm mit t_bad zutreffen. Diese Angriff wird für jede Möglichkeit der nächsten Stelle bis zur letzten Stelle druchgeführt ausgehend vor 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 Kombinatione 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 ein boolschen 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    }
```

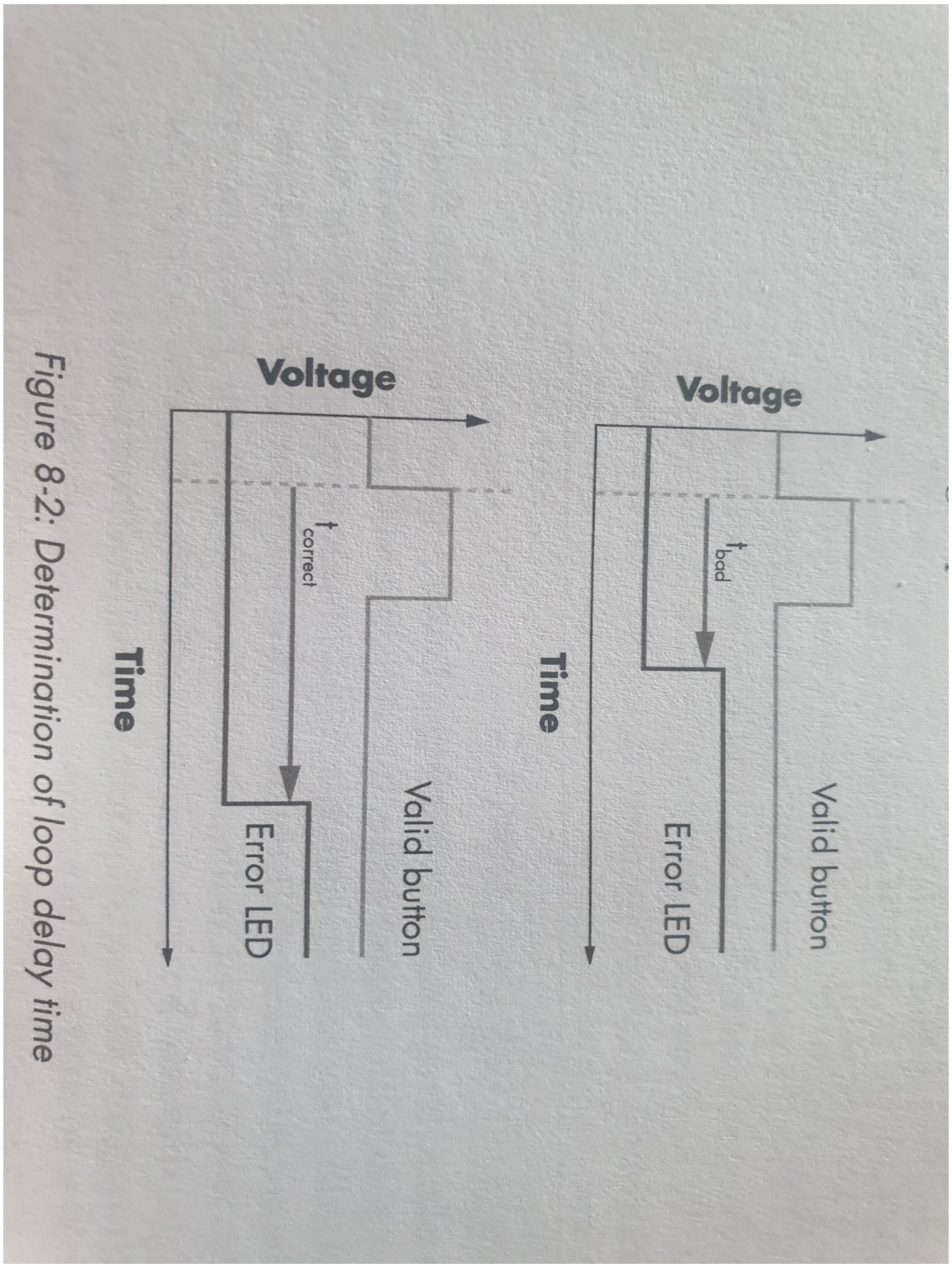

Figure 8-2: Determination of loop delay time

Abbildung 5: ARM Register Set

```
9       return test;
10    }
```

Eine ähnliche Schutzmaßnahmen, 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.

Auch basierend auf der Randomisierung könnte die Erweiterung helfen, den Start des Vergleichs zu randomisieren, d.h. bei jedem Vergleich wird eine andere Position als erste Stelle verglichen.

3.2. Format String Vulnerabilty

3.2.1. Beschreibung

Eine Format String Vulnerabilty 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 basiert auf variadische Funktionen, d.h. Funktionen, die eine variable Anzahl an Argumenten akzeptieren. Eine von jedem C Programmiere benutzere variadische Funktion ist beipsielsweisedie printf Funktion. Das erste Argument einer printf Funktion ist der sogenannte Format String, der mit den angegebenen Paramtern beginnend mit %, wie %s, %d, %x usw., besimmt, wie nachfolgende Parameter als Argumente verwendet werden. 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.

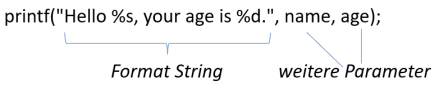

Abbildung 6: printf - Format String

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 werden. Allgemein funktiert dies, wenn der Format String nicht der Anzahl der enstprechenden nachfolgenden Argumente entspricht, weshalb dann unsichere Speicheroperationen, wie Zugriff auf Register bzw. Stack, durchgeführt werden.

```c
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 Progamm gewinnen. Dazu können beipsielsweise Passwörter zählen, die im Speicher abgespeichtert sind.

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 Vulnerabilty 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 Addresse zum Ausführen der Shell überschreiben.

3.2.2. Beispiel

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

```c
int main(){
  char* correct_password = "f0rm4tS7r!ng";
  char* username;
  int age;
  // User Input
  scanf("Hello, please first enter your age: ", &age);
  scanf("Enter your username: ", username);
  // For taking advantage of this vulnerability:
  // 1. enter a number for which adress to read,
  // 2. enter on the correct position %s to interpret the entered number in
  // Output
  printf("Your age is: %d", age);
  printf("Welcome ");
```

```
14    printf(username); // FORMAT STRING Vulnerability
15  }
```

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 auslzusende Speicherzelle dient. In der zweiten Eingabe, der Benutzername eingabe, kann man nun mit enstprechend 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 Vulnerabilty existiert eine einfache sehr effektive Schutzmaßnahme, nämlich das sichere Programmieren, indem man die Paramter %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. COMPILER BASED COUNTERMEASURES

3.3. Buffer Overflow und 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. Die Eingabe überschreitet die Grenzen des Buffers und überschreibt damit andere Daten, die außerhalb des Buffers gespeichert sind. Diese Schwachstelle kann in zwei verschiedenen Kontexten auftreten,

nämlich in einem Stack- oder in einem Heap Buffer Overflow. In diesem Kapitel wird der Fokus auf den Stack Buffer Overflow gelegt, wobei man im Wesentlichen zwei verschiedenen Angriffsarten unterscheidet. Ein Angriffstyp basiert auf der Idee eigenen Code einzufügen und diesen auszuführen, indem der Rücksprungadresse der Funktion so überschrieben wird, dass die Funktion mit eigens eingefügten Code ausgeführt wird. Diese Art von Angriff ist relativ leicht zu verhindern, indem eine Data Execution Prevention (DEP) verwendet wird. Wie im Kapitel 2 erwähnt, ist hierfür die MPU bei der STM Mikrocontroller Familie zuständig, der den SRAM und damit den sich dort befindenden Stack als nicht ausführbar markiert.

Die Verhinderung des zweiten Angriffstypes ist deutlich komplexer, denn hier werden vorhandene Funktionen zur Ausführung angebracht, indem die return Adresse mit der Adresse dieser gewünschten Funktion überschrieben wird. Dieser Angriffstyp ist als ROP Angriff bekannt, namensgebend durch die RETN Assembly Instruktion. Im folgenden Beispiel wird ein ROP Angriff mit Berücksichtigung der Eigenschaften des STM32 dargestellt.

3.3.2. Beispiel

Ein ROP Angriff funktioniert wie in der folgenden Abbildung dargestellt. Eine beispielhafte Funktion request_input() lässt dem User über eine weitere Funktion, scanf(), eine Eingabe tätigen, wobei das zugrundeliegende Speicherobjekt beispielsweise ein C Array ist. Damit der Angriff möglich ist, muss die Eingabe so gewählt werden, dass ein Buffer Overflow auftritt, d.h. mehr Daten in das Array gespeichert werden, als dieses umfasst. O.B.d.A wird hier ein char Array der Größe 8 gewählt. Bei der Eingabe befindet man sich in der scanf Funktion, die nun beispielsweise eine Eingabe der Länge 24 entgegennimmt: *AAAA AAAA AAAA AAAA AAAA 0ROP* (Leerzeichen nicht als Eingabe, nur zur Lesbarkeit) Für die mögliche Eingabe wurden auf dem nach unten wachsenden Stack (d.h. von hohen zu niedrigen Adressen) nur 8 Byte reserviert. Nun werden zu den höheren Adressen hin (der Stack wächst ja nach unten bei der Reservierung) weitere Daten überschrieben. Bei der vorhandenen ARM Architektur, die sich im STM32 befindet, befindet sich wie eingangs erwähnt ein LR Register, welches die Return Adresse der Scanf Funktion noch hält, da keine weiter Funktion aufgerufen wird. Deshalb funktioniert der Rücksprung von der scanf Funktion in die request_input() Funktion, denn hierbei wird nur das LR Register in den

PC geladen. Dies ist auch der Grund, wieso zwei Funktionen dafür benötigt, werden, wobei die eine in der anderen genestet sein muss.

Bei der request_input() Funktion ist auf dem Stack die return Adresse überschrieben worden, wobei das Überschriebene dann in das LR geladen wird. Dies wird daraufhin wieder in den PC geladen, in dem Fall *POR0*. Hier ist noch zu bemerken, dass das als Little Endian interpretiert wird, und somit letztendlich zu der Funktion an der Adresse *0x0ROP* gesprungen wird. Der Rücksprung der aufgerufenen Funktion wird i.d.Regel fehlschlagen und das Programm abbrechen, jedoch bleiben die mit der gewünschten aufgerufenen Funktion bestehen. Damit ist der ROP Angriff erfolgreich.

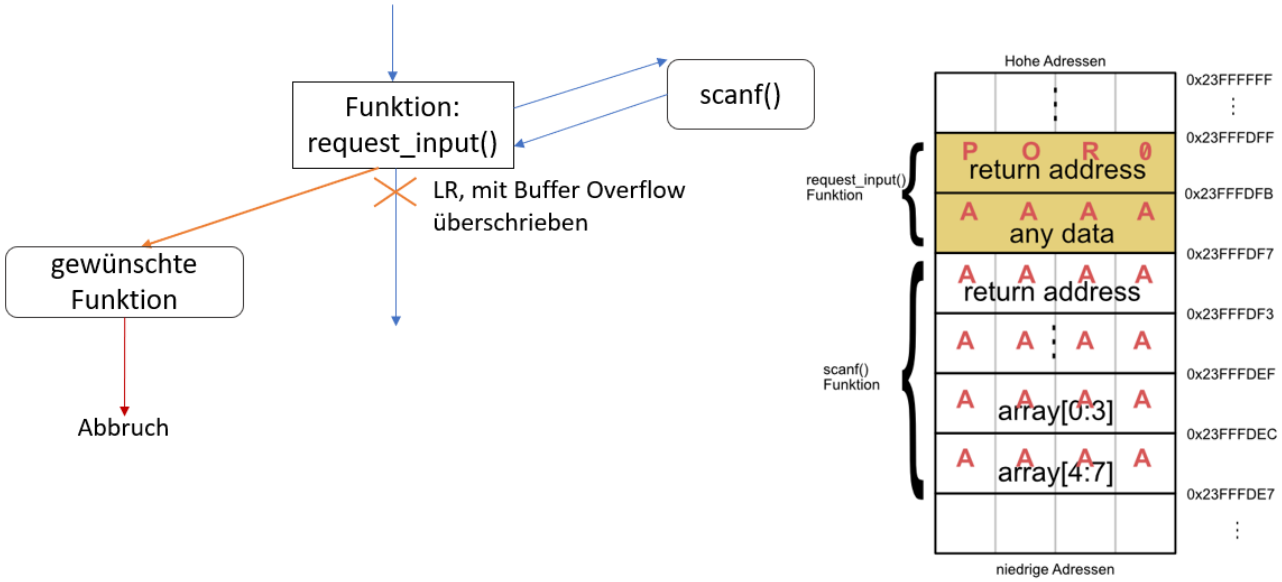

Abbildung 7: Buffer Overflow

3.3.3. Prävention/Schutzmaßnahmen

Neben der offensichtlichen und schwierigsten Schutzmaßnahme, keine Fehler in der Softwareentwicklung zu machen, gibt es eine Vielzahl von Schutzmaßnahmen. Eine davon, die DEP umgesetzt durch die MPU, ist bereits oben erwähnt, welche den Nachteil hat, dass diese den zweiten Angriffstyp ROP nicht verhindert.

Für diesen Angriffstyp gibt es andere Schutzmaßnahmen wie die Nutzung von Protection Rings, Adress Space Layout Randomization (ASLR), Stack Canaries und bzw. oder Watchdogs.

4. Besprechung der möglichen Skalierbarkeit

Die oben genannten Schwachstellen können auch in Infotainmentsystemen von Automobilen ausgenutzt werden mit dem Ziel die Kontrolle über dieses zu erlangen. Hier wird inbesondere die Ausnutzung der Format String Vulnerabilty betrachtet und wie diese im konkreten Ausnutzungsfall am Automobil skaliert werden kann.

Mittlerweile hat jeder größere Autombilhersteller seine eigenes Infotainmentsystem bzw. in Kooperation mit Softwareriesen wie Google, Apple, etc. entwickelt, und jedes Neufahrzeug bietet dieses auch an. Die Funktionalität der Infotainmentsysteme steigt deutlich an, von Nutzung eines Navigationssystem über Verbindung mit Mobilgeräten und Nutzung von weiteren Apps. Insbesondere Schnittstellen nach außen, wie die Bluetooth Kommunikation des Automobils mit einem Smartphone birgt einige Risiken. Darunter zählt auch die FOrmat String Vulnerability Schwachstelle. Diese kann man nämlich bei der Verbindung eines Mobilgerätes mit dem Infotainmentsystem über Bluetooth ausnuzten, indem der Name des mit sich zu verbindenden Smartphones beispielsweise auf *%c%c%c%c%c%c%c* geändert wird. Wie in der Format String Vulnerability erklärt, führt diese Eingabe bei unsicher programmierten printf statements dazu, dass Register bzw. der Stack ausgelesen werden. Dies war hier der Fall, und man konnte Daten auslesen, die von Experten als Karteninformationen identifiziert wurden.

Allein das Auslesen von wichtigen Informationen des Automobils ist eine erhebliche Sicherheitslücke, jedoch kann dies weiter skaliert werden mit der Nutzung von *%s* oder *%n*. Mit %n kann insbesondere nprintf() so ausgenutzt werden, dass ein Wert geschrieben wird an die Speicheradresse, die am Stack referenziert wird. Der Angreifer kann also eine beliebige Adresse eingeben, dementsprechend %n ausnutzen, um dan diese eingegebene Adresse einen Wert zu schreiben. Falls dies eine ausgesuchte Rücksprungadresse oder irgendeinen Pointer überschreibt kann der Angreifer nun entweder eigens eingegebenen Code ausführen oder bereits bestehende Funktionen ähnlich wie bei Buffer Overflow ROP Angriff ausführen.

Diese Schwachstelle kann also von einer Informationsgewinnung zur Ausührung bei geschickter Anwendung von %n und dementsprechend vorliegenden Schwachstellen führen. Bei vorliegender Schwachstelle kann der Anriff also so skaliert werden, dass anstatt nur Informationen ausgelesen werden, eigener

Code ausgeührt wird (Remote Code Execution), womit prinzipiell die Kontrolle über das Automibil erlangt werden kann. Dies wiedrum kann in verschiedensten Szenarien ausnutzen. Da die Steuergeräte (ECUs) im Auto meist untereinander verbunden sind, lässt sich nun auch ein Angriffsszenario vorstellen, bei dem über Remote Code Execution angefangen beim Informationssystem auf Lenk-/Bremsfunktionen mit erheblichen Auswirkungen manipuliert werden können.

A. Abbildungsverzeichnis

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