OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

STUDIENARBEIT

Gruber Daniel

Schwachstellen eingebetteter Systeme

19. Juli 2022

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

Inhaltsverzeichnis

Abkürzungsverzeichnis

ASLR	Address Space Layout Randomization
DEP	Data Execution Prevention
DOS	Denial of Service
LR	Linking Register
MMU	Memory Management Unit
MPU	Memory Protection Unit
PC	Program Counter
RISC	Reduced Instruction Set Computer
ROP	Return Orientated Programming

1. Einleitung

Embedded Systems befinden sich in vielen Gegenständen und Geräten unseres täglichen Lebens, auch beispielsweise im Fahrzeug. Die Vernetzung im Fahrzeug und damit auch die Anzahl von Mikrocontrollern nehmen im aktuellen Jahrzehnt deutlich zu, weshalb in Deutschland unter anderem die Automobilhersteller im Bezug auf Sicherheit des Fahrzeugs im Fokus stehen. Autombilhersteller wie BMW, AUDI und Daimler sind erst seit einigen Jahren vertieft 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, Buffer Overflows und Format String Vulnerabilities. [4] [6]

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 Schutzmaßnahmen 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 wird vom europäischen Halbleiterhersteller STMicroelectronics N.V. produziert, welche als eine der ersten Hersteller unter anderem die CORTEX M3 Lizenz von der Firma ARM erworben hat. Die STM32 Controllerfamilie zeichnet sich durch einen 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.

Der Cortex-M0/M3/M4 Prozessor, insbesondere der Cortex-M3 Prozessor wie beim STM32F103C8T6 vorhanden, basiert grundlegend auf der Harvard Architektur. Hierbei existieren, wie für die Harvard Architektur typisch, verschiedene Busse für Befehle und Daten, welche es ermöglichen, 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 Befehl- und Datenbus existiert, jedoch sowohl Befehle und Daten im gleichen Speichermedium, dem Flash, liegen und somit der Adressraum dementsprechend linear programmiert werden kann. Grundsätzlich wird dies in der Literatur als Modifizierte Harvard Architektur bezeichnet, da es zwar verschiedene Busse für Daten und Befehle gibt, jedoch keine strikte Trennung zwischen Daten und Befehlsadressraum gegeben ist. Insbesondere ist 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 sichert man sich den

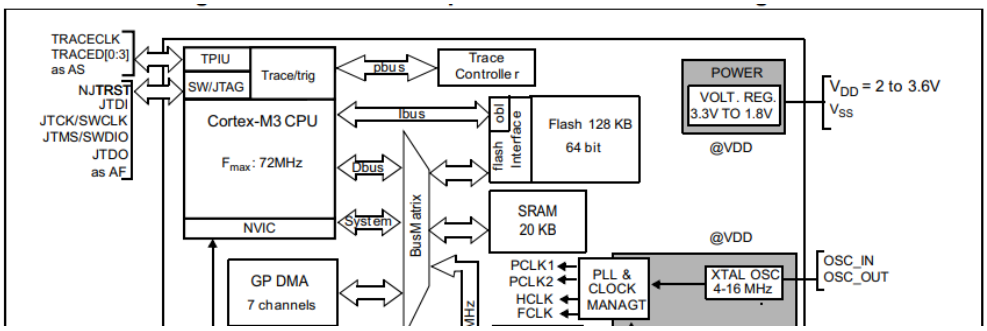


Abbildung 1: STM Architecture

Vorteil der Harvard-Architektur, dass gleichzetiges Laden von Befehlen und Daten für bessere Performance möglich ist, jedoch verliert man die Eigenschaft des pyhsikalisch getrennten Adress- bzw. Speicherbereichs. Hier wird wie in der Von-Neumann Archiktur Daten und Befehle auf dem gleichen Speichermedium abgespeichert, welche durch verschiedene Adressbereiche getrennt sind. Da dies aber keine getrennten physikalischen Speicherbereiche sind, können beispielsweise Schwachstellen wie Buffer Overflows und Return Orientated Programming ausgenutzt werden, um mit eigenen Daten den Programmfluss bzw. Befehle zu modifizieren. [14] [16] [17] [18]

Manche Architekturen bieten des Weiteren an, die Ausführung von Daten zu verhindern, indem eine Hardware unterstützte Memory Management Unit (MMU) oder Memory Protection Unit (MPU) entsprechend konfiguriert wird. Die MMU

besitzt in der Regel mehr Funktionalitäten als die MPU, welche sich nur auf Speicherschutz konzentriert. Da die STM32 Mikrocontroller Familie wenn dann nur eine MPU besitzt, wie z.B. der STM32F303, wird hier nur darauf eingegangen. Die MPU ermöglicht es, ein eingebettetes System robuster und sicher zu machen, indem beispielsweise der SRAM bzw. Bereiche vom SRAM als nicht-ausführbar definiert werden können und damit gewisse Schwachstellen nicht mehr ausgenutzt werden können. [11]

Das Speichermodell bzw. der Adressierungsbereich von möglichen 4GB der CPU ist in Abbildung 2 dargestellt. Teil dieses Adressierungsbereichs sind der Code, der sich im Flash befindet, und der SRAM, welche beide in Abbildung 3 dargestellt sind. Dabei ist insbesondere wichtig, dass der Stack nach unten, d.h. von höheren zu niedrigen Adressen wächst, was es ermöglicht, die Rücksprungadresse und andere Bereiche im SRAM über einen Buffer Overflow oder eine Format String Vulnerabilty für Angriffe zu überschreiben. [12]

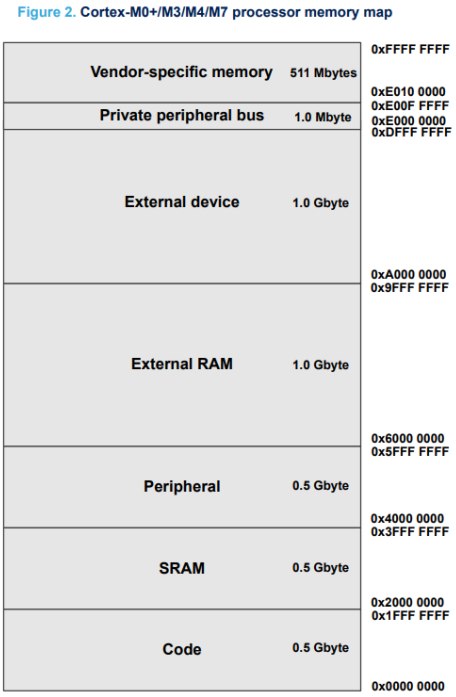


Abbildung 2: Memory map

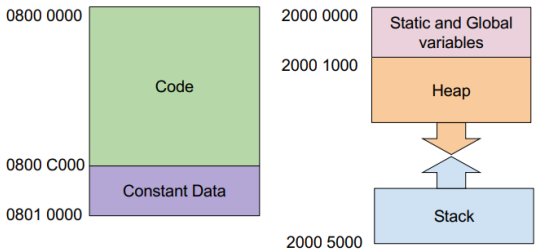


Abbildung 3: Flash und SRAM

Auch die Funktionsweise des Linking Register (LR) in Kombination mit dem Program Counter (PC) Register ist für die konkrete Ausnutzung nachfolgender Schwachstellen bedeutend. Denn das LR speichert die Rücksprungadresse der Funktion, welche nachfolgend dann in den PC geladen wird. Der PC gibt an, wo das Programm fortgesetzt wird. Erst wenn mehrere Funktionen vorhanden sind, wird die Rücksprungadresse auf dem Stack abgelegt und muss von diesem

wieder in das LR geladen werden, welches dann wiederum in den PC geladen wird. [14] Deswegen werden für das Überschreiben der Rücksprungadresse zwei Funktionen benötigt, wobei eine die andere aufruft, um das Überschreiben der Rücksprungadresse der äußeren Funktion zu ermöglichen.

Abbildung 4 zeigt die vorhandenen Register im STM32F103C8T6 auf, wobei neben den genannten Registern noch der Stack Pointer und die General Purpose Register zu erwähnen sind. [12]

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

Quelle: [12]

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. Aber auch bei der Format String Vulnerability muss beim Überschreiben der Adresse auf die Endianness geachtet werden. [12] [13]

Wie viele Embedded Systems haben die Mikrocontroller der STM32 Familie kein Betriebssstem, das weitere Schutzmechanismen oder ähnliche Dienste bieten würde.

3. Schwachstellen

3.1. memcmp Timing Attacke für Bruteforcing

3.1.1. Beschreibung

Die memcmp Timing Attacke ist ein typischer Seitenkanal-Angriff. Dieser Angriffstyp basiert auf Informationen, die von der konkreten Implementierung eines Systems abhängen. Bei der memcmp Timing Attacke basiert dies auf dem Wissen über die benötigte Zeit eines Vergleichs von Speicherbereichen, welcher in der Software implementiert ist. [1]

Vorausgesetzt eine Speichervegleichsfunktion ist so implementiert, dass beim ersten nicht übereinstimmendem verglichenen Zeichen von der Funktion *false* zurückgegeben wird, benötigt der Vergleich unterschiedlich lange, je nach Anzahl richtiger Zeichen einer Zeichenkette. Hier wird konkret der Aspekt der Zeit ausgenutzt, denn die Dauer der Funktion hängt von der Anzahl und Richtigkeit der zu vergleichenden Speicherbereichen bzw. Zeichenketten 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 reinem 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 Funktion für die jeweilige Stelle einen erfolgreichen Vergleich durchgeführt. Dies führt dazu, dass die nächste Stelle überprüft wird, was insofern bedeutet, dass die Funktion mehr Iterationen zu durchlaufen hat und damit mehr Zeit benötigt. 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 auszutesten sind. [1]

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 Bereich) als die Dauer des Vergleichs. Aus diesem Grund werden für solche memcmp Timing

Attacken Oszilloskope oder Logic Analyzer benötigt, um den Zeitunterschied für den Vergleich am Embedded System zu messen. [1]

3.1.2. Beispiel

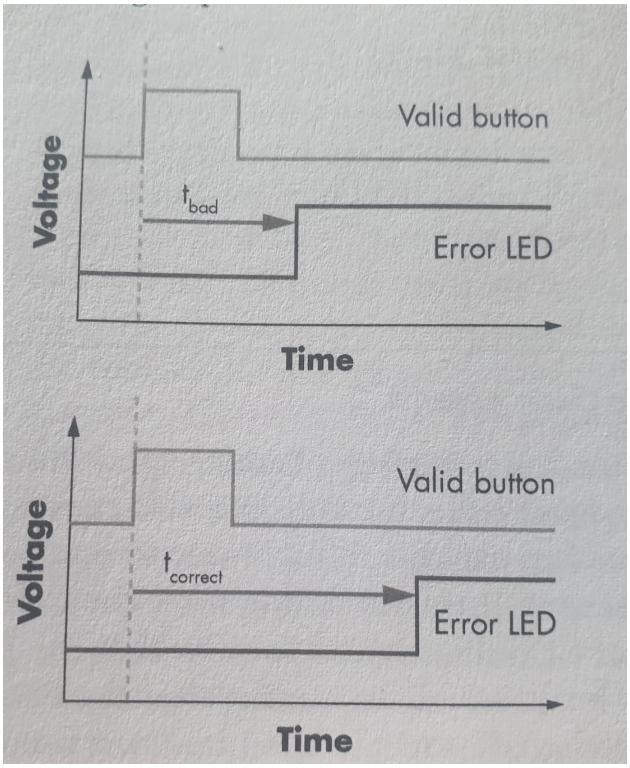
In diesem Abschnitt wird ein repräsentatives Beispiel für oben beschriebene Schwachstelle dargestellt. Der Einfachheit halber wird ein PIN Vergleich der begrenzten Länge 4 durchgeführt, wobei das Alphabet 0-9 ist, d.h. eine Mächtigkeit von $|A| = 10$ besitzt. Dies gilt ohne Einschränkung der Allgemeinheit und kann beliebig in der Länge sowie der Mächtigkeit des Alphabets verändert werden. Zudem wird die Annahme getroffen, dass der PIN Vergleich erst nach vollständiger PIN Eingabe erfolgt. Dabei wird folgende Methode für die Überprüfung der PIN 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  }
```

Abbildung 5: memcmp Timing Attacke - Methode zur Überprüfung einer PIN

Für reines Raten, d.h. Bruteforcing ohne weitere Kenntnisse, sind $10*10*10*10 = 10^4$ Kombinationen auszutesten. Um die Anzahl der Kombinationen deutlich zu reduzieren, kann man den Vorteil des Wissens über die Implementierung der oben dargestellten Funktion nutzen und damit das Prinzip der memcmp Timing Attacke verwenden. Denn die Methode für das Überpüfen der PIN gibt beim ersten nicht korrekten Zeichen 'false' zurück, weshalb die die Ausführungsdauer der Funktion von der Anzahl der richtig eingegebenen PIN Stellen abhängt. Dafür wird Stelle für Stelle durchgegangen und überprüft, angefangen bei der ersten Stelle für jede mögliche Eingabe von 0-9, welche Eingabe die längste Zeit benötigt hat. Denn wenn die Stelle richtig ist, war der Vergleich

richtig und die Funktion wird die nächste Stelle überprüfen, was mit einer längeren Dauer der Funktion verbunden ist. 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. Für die erste Eingabestelle stellt Abbildung 6 oben dar, wie sich die Vergleichszeit im korrekten Fall ($t_{correct}$) zum Fehlerfall (t_{bad}) unterscheidet. Da die 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



Quelle: [1]

Abbildung 6: Timing Attack

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 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. [1]

3.1.3. Prävention/Schutzmaßnahmen

Für die dargestelle Funktion in Abbildung 5 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 vier 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 erst dann das Ergebnis des Vergleichs zurückgegeben wird. Hierbei wäre für oben dargestellten Code genau eine Änderung nötig, nämlich die Verwendung einer 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. Die dafür notwendige Änderung ist in Abbildung 7, insbesondere in Zeile 3, dargestellt.

```
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  }
```

Abbildung 7: memcmp Timing Attacke - verbesserte Methode zur Überprüfung einer PIN

Eine ähnliche Schutzmaßnahmen, die zwar nicht auf konstante Zeit setzt, sondern auf Randomisierung von Zeit, kann durch Hinzufügen von zufälligen Operationen jeglicher Art, z.B. *sleep()* Statements, 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 jeder neuen Eingabe wird der Vergleich bei einer anderen Position startend durchgeführt.

Eine weitere Schutzmaßnahme ist die Ausführung von *Decoy Operations* (dt. Ablenkungsmanöver), wobei Algorithmen zufällige Berechnungen durchführen. Dabei werden beispielsweise manche Werte zufällig öfter verglichen oder die Ergebnisse von zufälligen weiteren Berechnungen nicht verwendet. [3]

Alle vorgestellten Maßnahmen zielen darauf ab, Seitenkanalinformationen zu verfälschen bzw. damit die Nachvollziehbarkeit des Algorithmus zu erschweren.

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 zu bringen. Diese Schwachstelle basiert auf variadischen Funktionen, d.h. Funktionen, die eine variable Anzahl an Argumenten akzeptieren. Eine von jedem C Programmierer benutzte variadische Funktion ist beispielsweise die *printf* Funktion. [10] Das erste Argument einer printf Funktion ist der sogenannte Format String, der mit den angegebenen Parametern beginnend mit %, wie %s, %d, %x usw., besimmt, wie nachfolgende Parameter als Argumente verwendet werden. Abbildung 8 verdeutlicht dies, wobei *name* ein string und *age* eine integer Variable ist, die in den entsprechenden Parametern %s und %d als Argumente ersetzt werden.

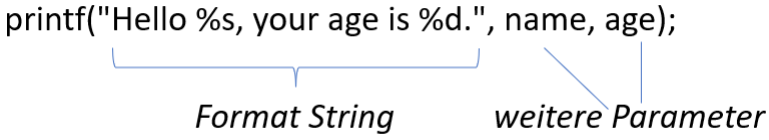


Abbildung 8: printf - Format String

Falls die printf - Funktion unsicher programmiert ist, wie in Abbildung 9 zu sehen, wird diese Funktion ohne explizite Parameter aufgerufen, weshalb die Werte von den Registern bzw. weiterführend vom Stack ausgelesen werden. Allgemein funktioniert dies, wenn der Format String nicht der Anzahl der nachfolgenden Argumenten entspricht, weshalb dann unsichere Speicheroperationen, wie Zugriffe auf die Register oder den Stack durchgeführt werden. Durch das

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

Abbildung 9: Format String Read Vulnerability - Beispiel

Auslesen von Registern, Stack und Speicher kann ein Angreifer wertvolle Informationen über das laufende Progamm gewinnen. Dazu können beispielsweise Passwörter zählen, die im Arbeitsspeicher (RAM) oder in anderen Speicherbereichen vorzufinden sind.

Des Weiteren ist es möglich, wie anfangs erwähnt, eigenen eingegebenen Code auszuführen. Dabei ist oft das Ziel, eine (Admin-) Shell zu öffnen, auf der weitere Aktionen ausgeführt werden können. Zuerst wird dabei der Shell Start in Assembly gesucht. Mit der Format String Vulnerabilty wird darauffolgend mit *%x* die Rücksprungadresse der jeweiligen *print* Funktion durch Ausprobieren herausgefunden. Nachfolgend kann die Schwachstelle ausgenutzt werden, indem im Input die Adresse der Rücksprungadresse von printf geschrieben wird und daraufhin %n ausgenutzt wird, um an diese Adresse die Adresse zum Ausführen der Shell zu schreiben. [15]

3.2.2. Beispiel

Folgendes Beispiel konzentriert sich auf die beispielhafte Ausnutzung der Format String Vulnerabilty in Bezug auf das Auslesen von Speicher. Der Codeausschnitt in Abbildung 10 zeigt ein Programm, dass oben beschriebene Schwachstelle in der *printf* Funktion aufweist. Hierbei wird der Nutzer nach zwei Eingaben gefragt, nämlich dem Alter und dem gewünschten Benutzernamen. Letztere Eingabe weist die entsprechende Format String Schwachstelle auf. Dies

```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
  //    entered number in 1. as pointer
  // Output
  printf("Your age is: %d", age);
  printf("Welcome ");
  printf(username); // FORMAT STRING Vulnerability
}
```

Abbildung 10: Format String Write Vulnerability - Beispiel

kann der Nutzer bzw. der Angreifer ausnutzen, indem er als Alter eine beliebige Zahl eingibt, die als Adresse für die auszulesende Speicherzelle dient. In der zweiten Eingabe, der Eingabe für den Benutzernamen, kann nun mit entsprechend vielen *%d* gefolgt von einem *%s*, welches sich genau an der Position des vorher eingegebenen Alters, diese Schwachstelle ausgenutzt werden. Beispielsweise kann die Eingabe 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. Damit kann beispielsweise nun der RAM, der Flash oder andere Speicherbereiche ausgelesen werden.

3.2.3. Prävention/Schutzmaßnahmen

Für die Format String Vulnerabilty existiert eine einfache sehr effektive Schutzmaßnahme, nämlich das sichere Programmieren, indem die Parameter wie *%s*, *%d* oder weitere korrekt benutzt werden. Dabei wird man zusätzlich von sicheren *print* Funktionen unterstützt, wie z.B. *sprintf*. Weitere Schutzmaßnahmen in der Software könnten zudem noch sein, die Eingabe des Nutzers zu überprüfen, und derartige möglicherweise schadhafte Eingaben nicht zuzulassen.

Des Weiteren gibt es Schutzmaßnahmen, die in den Compilern implementiert sind, sodass die Compiler mindestens eine Warnung oder sogar einen Fehler bei der Programmierung von unsicherem Code insbesondere in Zusammenhang mit variadischen Funktionen anzeigen. [10]

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. [8] Die Eingabe überschreitet die Grenzen des Buffers und überschreibt damit andere Daten, die außerhalb des Buffers gespeichert sind. Diese Schwachstelle kann in zwei verschiedenen Szenarien 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. [5] Wie in Abschnitt 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 deklariert.

Die Verhinderung des zweiten Angriffstyps ist deutlich komplexer, denn hier werden vorhandene Funktionen zur Ausführung gebracht, indem die Rücksprungadresse mit der Adresse dieser gewünschten Funktion überschrieben wird. Dieser Angriffstyp ist als Return Orientated Programming (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 Abbildung 11 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. es wird versucht, mehr Daten in ein Array zu speichern, als dieses umfasst. Ohne Beschränkung der Allgemeinheit 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 nur 8 Byte reserviert. Nun werden zu den höheren Adressen im Stack hin weitere Daten überschrieben. Bei der vorhandenen ARM Architektur, auf die die STM32 Mikrocontroller Familie basiert, befindet sich, wie eingangs erwähnt, ein LR, welches noch die Rücksprungadresse der *scanf* Funktion hält, da keine weitere Funktion aufgerufen wird. Deshalb funktioniert der Rücksprung von der *scanf* Funktion in die *request_input()* Funktion, denn hierbei wird nur das LR in den PC geladen. Dies ist auch der Grund, wieso zwei Funktionen dafür benötigt werden, wobei diese ineinander verschachtelt sein müssen.

Bei der *request_input()* Funktion ist auf dem Stack die Rücksprungadresse überschrieben worden, wobei das Überschriebene dann in das LR geladen wird. Dies wird daraufhin wieder in den PC geladen, in diesem Fall *POR0* wie in Abbildung 11 zu sehen. 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 in der Regel fehlschlagen und das Programm abbrechen, jedoch bleibt das gewünschte Ergebnis mit der aufgerufenen Funktion bestehen. Damit ist der ROP Angriff erfolgreich. [5] [9]

3.3.3. Prävention/Schutzmaßnahmen

Neben der offensichtlichen und vermeintlich schwierigsten Schutzmaßnahme, keine Fehler in der Softwareentwicklung zu machen, gibt es eine Vielzahl von Schutzmaßnahmen. Eine davon, die DEP umgesetzt durch die MPU, bereits in Abschnitt 2 ausführlich erklärt, deklariert Speicherbereiche im SRAM bzw. Stack als nicht ausführbar und verhindert damit den ersten Angriffstyp. Jedoch wird der zweite Angriffstyp ROP nicht verhindert, da dieser auf bereits existierende Funktionen zugreift, die nicht im Stack liegen.

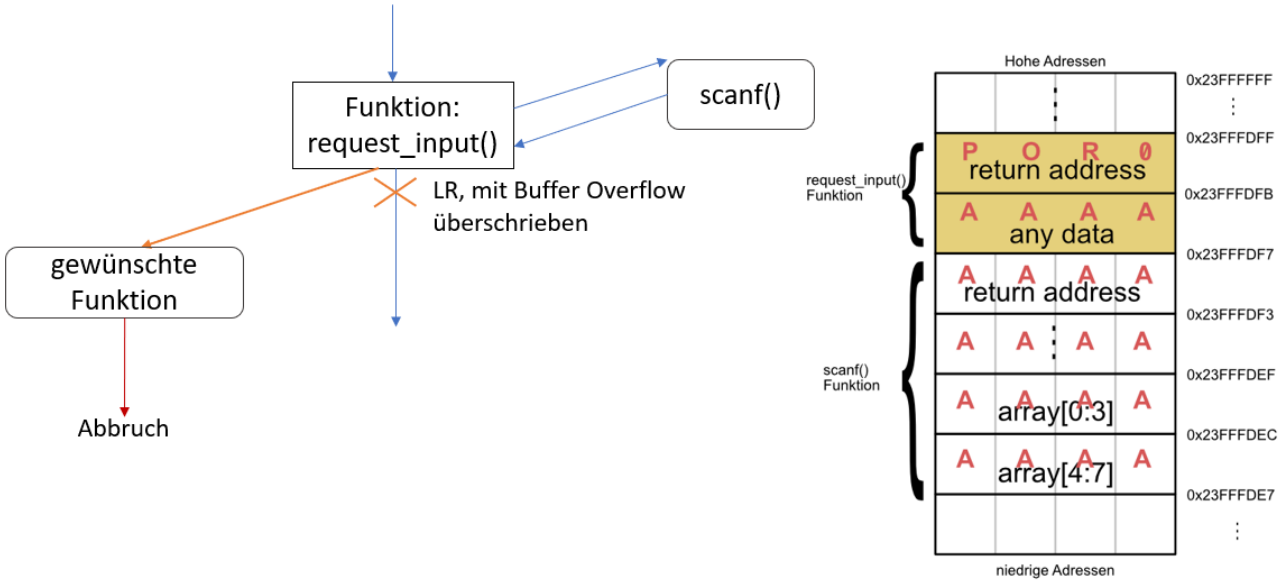


Abbildung 11: Buffer Overflow

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

Das *Protection Ring* Modell, welches von der Hardware unterstützt werden muss, dient der Trennung von Privilegien, mit denen verschiedene Prozesse laufen. Die grundlegende Idee hinter diesem Modell ist, dass Prozesse nach dem Need-to-Know Prinzip nur mit den benötigten Rechten ausgeführt werden. Dabei werden in diesem Modell die Ausführungsrechte bzw. -privilegien als eine Menge von Ringen beschrieben, wobei diese hierarchisch von innen nach außen nummeriert werden. Der innere Ring, Ring 0, bietet alle Rechte und Privilegien, während weiter außen liegende Ringe nur einen Teil der Privilegien des nächstinneren Ringes zur Verfügung stellen. Üblicherweise wird dieses Modell so verwendet, dass das System mit den höchsten Privilegien gestartet und initalisiert wird, während Benutzer- und Applikationsprozesse mit niedrigeren Privilegien laufen. Damit wird erreicht, dass Applikationen mit einer Format String Schwachstelle nur noch soweit ausgenutzt werden, was die Privilegien, mit denen diese Applikation läuft, erlauben. In der Regel wird dies soweit eingeschränkt, dass man einen Angriff verhindert oder zumindest dessen schadhafte Auswirkungen reduziert. [2]

Eine weitere Möglichkeit, ROP Angriffe zu verhindern, ist die Verwendung der

ASLR, indem die Adressbereiche des Stacks, des Heaps und mögliche verlinkte Bibliotheken eines Programms unvorhersehbar und zufällig verteilt werden. Da bei Angriffen es oft wichtig ist, die exakten Speicheradressen herauszufinden, ist diese Methode sehr effektiv, um die Ausnutzung von Schwachstellen durch gezieltes Springen zu einen Programm deutlich zu erschweren. Eine detailliertere Umsetzung basiert beispielsweise darauf, nach definierten Zeitintervallen die Adressbereiche neu zu vergeben. [8] [9]

Des Weiteren gibt es eine Compiler basierte Lösung, nämlich unter anderem *StackGuard*, bei der ein Sicherheitsmechanismus im Compiler eingebaut ist, der korrupte bzw. modifizierte Rücksprungadressen überprüft und damit zumindest den Sprung dorthin verhindert. Dabei wird ein *Stack Canary* - ein zufälliger bekannter Wert - im Stack neben der Rücksprungadresse geschrieben, während eine Kopie davon in einem General Purpose Register r0-r12 (siehe Abbildung 4) gespeichert wird. Beim Abbau der Funktion und damit implizit des Stacks wird beim Rücksprung der *Stack Canary* auf dem Stack mit der Kopie verglichen, um zu überprüfen, ob diese bei einem möglichen Angriff manipuliert worden ist. Eine ähnliche, sogar verbesserte Version, bietet der *Stackshield* unter Linux. Dieser bietet eine Speicherung der Rücksprungadresse an einem weiteren Platz, welche als letzte Operation vor Beenden der Funktion dann wieder zurückgeschrieben wird. [5] [7]

Eine weitere Schutzmaßnahme sind *Watchdogs*, die wiederum in Software- und Hardware Watchdog unterteilt werden können. Diese funktionieren wie ein Timer Interrupt, welcher den Prozessor neustarten würde, falls ein solcher auftritt. Bei Software Watchdogs werden vom Programm periodisch Nachrichten gesendet, um den Timer zurückzusetzen, bei Hardware Watchdogs geschieht dies ähnlich, wobei dieser als physikalische Hardware mit dem Prozessor verbunden ist. Sollten also in einem Programm Schwachstellen ausgenutzt werden, können die Watchdogs aufgrund nicht zurückgesetzten Timer, den Prozessor neustarten und damit die Ausnutzung vorerst verhindern. [19]

Zusammengefasst gibt es eine umfassende Menge von Schutzmaßnahmen, von denen hier einige dargestellt wurden. Anzumerken ist jedoch, dass diese Schutzmaßnahmen oft nur verhindern, dass schadhafter Code ausgeführt wird, aber nicht, dass das System zum Absturz gebracht wird. Damit kann also die Verfügbarkeit des Systems beeinträchtigt werden, was auch als Denial of Service (DOS) bekannt ist.

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. Im Folgenden wird insbesondere die Ausnutzung der Format String Vulnerabilty im Infotainmentsystem eines Automobils betrachtet und des Weiteren auf mögliche Skalierungen eingegangen.

Mittlerweile entwickelt jeder größere Automobilhersteller sein eigenes Infotainmentsystem bzw. in Kooperation mit Softwareriesen wie Google, Apple, etc.. Die Funktionalität der Infotainmentsysteme steigt zunehmend an, von Nutzung eines Navigationssystem über die Verbindung mit Mobilgeräten und Nutzung von weiteren Applikationen. 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 konnte nämlich bei der Verbindung eines Mobilgerätes mit dem Infotainmentsystem über Bluetooth ausgenutzt werden, indem der Name des mit sich zu verbindenden Smartphones beispielsweise auf *%c%c%c%c%c%c%c%c* geändert wird. Wie in der Format String Vulnerability erklärt, führt diese Eingabe bei unsicher programmierten *printf* Implementierungen dazu, dass Register bzw. der Stack ausgelesen werden. Dies war in früheren Versionen des Infotainmentsystems möglich, und Daten, die von Experten als Karten- und Navigationsinformationen identifiziert worden sind, konnten ausgelesen werden.

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()* in folgender Art und Weise ausgenutzt werden, dass ein Wert an die Speicheradresse geschrieben wird, die am Stack referenziert wird. Der Angreifer kann also eine beliebige Adresse eingeben, dementsprechend *%n* ausnutzen, um dann an vorher 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 beim Buffer Overflow ROP Angriff ausführen.

Diese Schwachstelle kann also von einer Informationsgewinnung zur Ausführung von eigenem Code 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 ausgeführt wird (Remote Code Execution), womit prinzipiell die Kontrolle über das Automibil erlangt werden kann. Dies kann zu verschiedenen Angriffsszenarien führen. Ein Worst Case Szenario könnte sein, dass mit der Erlangung der Kontrolle über das Fahrzeug über Remote Code Execution angefangen beim Informationssystem Lenk- und Bremsfunktionen manipuliert werden, die zu erheblichen negativen Auswirkungen führen können. [4]

A. Abbildungsverzeichnis

B. Literatur

[1] Woudenberg, Jasper van and O'Flynn, Colin: *The Hardware Hacking Handbook*, San Francisco, CA, USA: No Starch Press, 2022.

[2] Silberschatz, Abraham, Peter B. Galvin, and Greg Gagne: *Operating System Concepts*, Wiley, 30. Jun. 2021.

[3] Woudenberg, Jasper van and O'Flynn, Colin: *Chapter 14: Think of the Children: Countermeasures, Certifications and Goodbytes*, n.d.. Zuletzt aufgerufen am 09.07.2022. [Online]. https://github.com/HardwareHackingHandbook/notebooks/blob/main/labs/HHH_14_Think_

[4] Automotive Security, *Automotive Security*. n.d.. Zuletzt aufgerufen am: 25.06.2022. [Online]. https://cydrill.com/cyber-security/automotive-security-how-to-brick-your-car/

[5] Prandini Marco und Ramilli Marco: *Return-Orientated Programming*. 10. Dez, 2012. Zuletzt aufgerufen am: 09.07.2022. [Online]. https://ieeexplore.ieee.org/document/6375725

[6] Ondrej Burkacky: *Automotive software and electronics 2030*. 2017. Zuletzt aufgerufen am: 09.07.2022. [Online]. https://www.mckinsey.com/ /media/mckinsey/industries/automotive%20and%20assembly software-and-electronics-2030-final.pdf

[7] J.P. McGregor, D.K. Karig, R.B. Lee: *A processor architecture defense against buffer overflow attacks*, 8. März 2004. Zuletzt aufgerufen am: 11.07.2022. [Online]. https://ieeexplore.ieee.org/document/1270612

[8] OWASP: *OWASP Buffer Overflow*. n.d.. Zuletzt aufgerufen am: 09.07.2022. [Online]. https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

[9] OWASP: *OWASP Buffer Overflow Attack*. n.d.. Zuletzt aufgerufen am: 09.07.2022. [Online]. https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

[10] OWASP: *OWASP Format String Vuln.*. n.d.. Zuletzt aufgerufen am: 09.07.2022. [Online]. https://owasp.org/www-community/attacks/Format_string_attack

[11] STM: *Managing memory protection unit in STM32 MCUs*. n.d.. Zuletzt aufgerufen am: 09.07.2022. [Online]. https://www.st.com/resource/en/application_note/dm00272912-managing-memory-protection-unit-in-stm32-mcus-stmicroelectronics.pdf

[12] ST: *STM32F103Cx8 u STM32F103xB DataSheets*. n.d.. Zuletzt aufgerufen am: 09.07.2022. [Online]. https://www.st.com/content/ccc/resource/technical/document/datasheet/33/d4/6f/1d/df/0

[13] ST: *STM32F10xxx/20xxx/21xxx/L1xxx3Cx8 DataSheets*. n.d.. Zuletzt aufgerufen am: 09.07.2022. [Online]. https://www.st.com/resource/en/programming_manual/pm0056-stm32f10xxx20xxx21xxxl1xxxx-cortexm3-programming-manual-stmicroelectronics.pdf

[14] Bruce Hemingway: *ARM Overview*. 2017. Zuletzt aufgerufen am: 09.07.2022. [Online]. https://courses.cs.washington.edu/courses/cse474/17wi/pdfs/lectures/03-arm_overview.pdf (ARM)

[15] Code Arcana: *Introduction to format string exploits*. 02. Mai 2013. Zuletzt aufgerufen am: 09.07.2022. [Online]. https://codearcana.com/posts/2013/05/02/introduction-to-format-string-exploits.html

[16] Mikrocontroller.net: *Prozessorarchitekturen*. 25. Sep. 2018. Zuletzt aufgerufen am: 18.07.2022. [Online]. https://www.mikrocontroller.net/articles/Prozessorarchitekturen

[17] Mikrocontroller.net: *STM32*. 21. Jan. 2020. Zuletzt aufgerufen am: 18.07.2022. [Online]. https://www.mikrocontroller.net/articles/STM32

[18] Kadah, Yassar M.: *STM32 Microcontroller Lecture*. n.d.. Zuletzt aufgerufen am: 18.07.2022. [Online]. https://www.k-space.org/Class_Info/STM32_Lec2.pdf

[19] ST: *STM32 Watchdogs*. Sep. 2021. Zuletzt aufgerufen am: 18.07.2022. [Online]. https://www.st.com/content/ccc/resource/training/technical/product_training/
WDG_TIMERS-Independent-Watchdog-IWDG/files/STM32WB-WDG_TIMERS-Independent-Watchdog-IWDG.pdf/jcr:content/translations/en.STM32WB-WDG_TIMERS-Independent-Watchdog-IWDG.pdf

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:	03. Juni 2022 – 20. Juli 2022
Betreuung:	Schmidt Jonas