OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

STUDIENARBEIT

Gruber Daniel

Vulnerabilty eingebetteter Systeme

19. Juni 2022

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

Inhaltsverzeichnis

Abkürzungsverzeichnis

ECU Electronic Control Unit

1. Einleitung

Die Vernetzung im Fahrzeug sowohl mit dem Internet als auch intern nimmt immer weiter zu. Die Anzahl der ECU ist von einer kleinen unabhängigen Architektur zu eine funktionsspezifischen und weiterhin zu einer zentralisierten Architektur gewachsen mit bis zu circa 150ECUS einem Premium Segement Fahrzeug. Mit dem starken Anstieg von ECU im Auto nimmt einerseits die Funktionialität für den Nutzer zu, jedoch bilden sich durch mehr Software und Funktionialität mehr Angriffsmöglichkeiten auf das System Äutoän sich. Hier ist es neben sicherem Code schreiben (Software) auch ein sehr wichtiger Aspekt die Hardware zu kennen und insbesondere deren Schwachstellen. In dieser Arbeit wird auf die STM32 Architektur eingegangen und anhand dieser drei typische Schwachstellen erklärt. Zudem wird zu jeder dieser Schwachstellen Präventionsmaßnahmen beschrieben. Abschließend wird die Skalierbarkeit einer dieser Schwachstellen in Form eines xxxxxx Angnriffes auf ein Fahrzeug aufgegriffen und darüber eine Diskussion dargestellt, welche Maßnahmen dagegen getroffen werden können und welche Vorteile/Nachteile diese Maßnahmen haben.

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 haben. Der STM32 Controller zeichnet sich nämlich durch eine 32-Bit ARM Cortex-M0/M3/M4 CPU aus, die speziell für Mikrocontroller neu entwickelt wurde. Die Cortex M3 wird inoffziell als leistungsfähigerer Nachfolger der ARM7 TDMI Controller betrachtet.

Diese ARchitektur verwendet ausschließlich den THumb2 Befehlssatz. Hauptbestandteil des Cortex M3 Prozessor, wie konkret 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 oft von eine Adeptive Harvard

Architektur, da es zwar verschiedene Busse für Daten und Befehle gibt, jedoch keine strikte Trennung zwischen Daten und Befehlsadressraum gegeben ist sowie kein getrennter physikalischer Speicher für Daten und Befehle verwendet wird. Dabei 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.

Der Speicherzugriff funktioniert wie auf folgender Abilldung dargestellt:

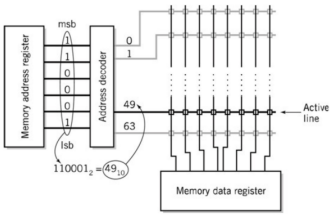

Abbildung 1: Speicherzugriff

Im Speicherzugriffsregister (Memory address register MAR) wird die Adresse angelegt, auf die im Speicher zugegriffen werden soll. Anschließen wird im Adress Decode die Adresse entschlüsselt/übersetzt und die jewelige Addresslinie wird aktiv geschalten. Daraufhin wird die aktiv geschaltene Linie, mit 8 indiviuellen Speicherzellen (jeweils 1 Bit), also insgesamt 1 Byte in das Speicherregistier geladen (Memory data register MDR).

Des Weiteren ist hier zu erwähnen, dass die STM32 Mikrocontroller-Familie auf Little Endian setzt, d.h. das niederwertigste Byte befindet sich an der niedrigsten Adresse. Hier wird beispielsweise ein integer ivar=0x01234567 folgendermaßen abgespeichert:

Abbildung 2: Little Endian

Die Abspeicherung in Little Endian spielt insbesondere für die Schwachstelle Buffer Overflow eine wichtige ROlle, da beim Auslesen des Speicher dies zu berücksichtigen ist.

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 Speichervegleichsfunktion so implementiert ist, dass beim ersten nicht übereinstimmende vergleichenenen 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 durchrpobiert 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 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 $|A|+|A|+|A|+|A|+|A|+|A| = 6*|A|$ 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 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 genannte 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 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 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. 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    }
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.

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 kann man beipsielsweise in printf, fprintf und weiteren print Funktionen ausnutzen. Das erste Argument ist der sogenannte Format String und die im Format String enthaltenen weiteren Parameter 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.

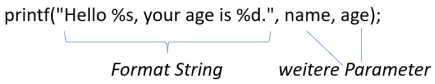

Abbildung 3: printf - Format String

Falls die printf - Funktion unsicher programmiert ist, wie auf folgender Abbiludng zu sehen, kann der Nutzer/Angreifer als Eingabe mehrere %x wie folgt eingeben: %x%x%x%x%x Da die printf Aufruf ohne explizite Parameter aufgerufen wird, werden die Parameter von den Registern bzw. weiterführend vom Stack ausgelesen und als Hexadezimal zurückgegeben. 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.

Des Weiteren ist ees möglich, wie anfangs erwähnt, eigenen eingegebenen Code auszuführen. Dabei ist oft das Ziel, ein Shell zu öffnen, auf der weitere Aktionen ausgeführt werden können.

3.2.2. Beispiel

Folgendes einfaches Beispiel zeigt die Verwundbarkeit von printf bei falscher Benutzung auf:

```
int main(){
  char* input = "Hello World!";
  printf(input);
}
```

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.

3.3. Buffer Overflow (ROP)

3.3.1. Beschreibung

3.3.2. Beispiel

3.3.3. Prävention/Schutzmaßnahmen

4. Besprechung der möglichen Skalierbarkeit

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:	Jonas Schmidt