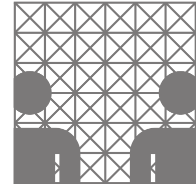
Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften



SVS-Masterprojekt IT-Sicherheit

Buffer-Overflow

Konstantin Schleser (9schlese@informatik.uni-hamburg.de)
Tim Krämer (7kraemer@informatik.uni-hamburg.de)
Denis Graf (7graf@informatik.uni-hamburg.de)

Betreuer: Dipl.-Wirtsch.-Inf. Dominik Herrmann & Karl-Peter Fuchs, M.A.

Inhaltsverzeichnis

1 Buffer-Overflows **1**
1.1 Introduction . . . 1
1.2 Allgemeines über Buffer Overflows . . . 1
1.3 Historische Angriffe . . . 2
1.4 Speichermanipulation . . . 3
1.5 Reverse Engineering . . . 4
1.6 Prozess und Speicherorganisation . . . 4
1.7 Bezug zu C . . . 7
1.7.1 C Strings . . . 8
1.7.2 Aufrufkonventionen . . . 8
1.8 Typen von Buffer-Overflows . . . 8
1.8.1 Stack Überlauf . . . 8
1.8.2 Heap . . . 9
1.9 Shellcode . . . 10
1.9.1 Null byte Problem . . . 10
1.10 x86 Architektur . . . 11
1.11 Assembler . . . 12
1.12 Shellcode Beispiel . . . 13
1.13 Sicherheitsmaßnahmen . . . 14
1.13.1 Canaries . . . 14
1.13.2 Address Space Layout Randomization . . . 15

2 Aufgaben **17**
2.1 Warm werden mit Hexadezimalzahlen . . . 17
2.2 Hauptaufgaben . . . 17
2.3 Zusatzaufgaben . . . 18
2.4 Hinweise und Hilfestellungen . . . 19
2.4.1 Eclipse . . . 19
2.4.2 Interactive Disassembler (IDA) . . . 21
2.4.3 Assembler . . . 21
2.4.4 ASCII-Tabelle . . . 23

3 Musterlösungen **25**
3.1 Lösungen zu Hauptaufgaben . . . 25
3.1.1 Ursprünglicher C-Code . . . 25
3.1.2 Analyse des Assembler-Codes . . . 26
3.1.3 Lösungen zu Aufgabe 1 . . . 31

3.1.4 Lösungen zu Aufgabe 2 32
3.1.5 Lösungen zu Aufgabe 3 34
3.1.6 Lösungen zu Aufgabe 4 36
3.2 Lösungen zu Zusatzaufgaben 36
3.2.1 Lösungen zu Aufgabe 1 36
3.2.2 Lösungen zu Aufgabe 2 36
3.2.3 Lösungen zu Aufgabe 3 37
3.2.4 Lösungen zu Aufgabe 4 37

Literaturverzeichnis **i**

1 Buffer-Overflows

1.1 Introduction

Pufferüberläufe (englisch buffer overflows), auch bekannt als Buffer Overruns stellten die häufigste Form von Sicherheitslücken in den zehn Jahren dar. Vorherrschend im Bereich der entfernten Angriffe auf Netzwerke (englisch remote exploits) werden Sie von anonymen Internetbenutzern ausgenutzt um Zugriff und Kontrolle auf verwundbaren Server zu erlangen. Sie treten dann auf, wenn ein Programm bei der Verarbeitung von Informationen aus externen Quellen – wie beispielsweise Benutzereingaben – Puffer mit einer statischen Größe anlegen und dann ohne eine explizite Längenüberprüfung die externen Daten in den Puffer schreibt. „It's common practice for programmers to pick an arbitrary large number of bytes for a buffer, and assume that no one will ever need more than that.“[6].

Wäre es möglich Buffer-Overflow Sicherheitslücken effektiv zu beseitigen, könnte ein Großteil der schwerwiegendsten Sicherheitsbedrohungen in der IT ebenfalls aufgelöst werden. Wir untersuchen die Entstehung von Buffer-Overflows und zeigen Möglichkeiten wie Schwachstellen in bestehender Software identifiziert werden können. Dafür befassen wir uns mit Software Reverse Engineering, dem Programmspeicher und der Programmierung in Assembler.

1.2 Allgemeines über Buffer Overflows

Bei Pufferüberläufen (englisch buffer overflows) handelt es sich um die Reaktion des Betriebsystems auf einen Fehler in bestehender Software oder bei der Ausführung dieser. Buffer-Overflows sind im letzten Jahrzehnt für die meisten Sicherheitlücken in Programmabläufen auf Online- und Offlinesystemen verantwortlich. Oft entstehen Programmierfehler, die zu Buffer-Overflows führen durch Unaufmerksamkeit beim Programmieren mit wenig abstrahierten Sprachen, wie C oder C++. Diese Sprachen werden nahezu direkt zu Maschinencode kompiliert und laufen im Gegenteil zu hoch abstrahierten Sprachen, wie Java oder Python durch keine, oder allenfalls rudimentären Kontrollstrukturen des Betriebsystems. Bei Buffer-Overflows handelt es sich um den Fehler der ermöglicht, dass zu große Daten in einen nicht ausreichend dimensionierten Puffer des Arbeitsspeichers des Betriebsystems gelangen und diesen Puffer damit überlaufen lassen. Dieses Fehlverhalten hat zur Folge, dass Speicher anderer Funktionen des ausgeführten Programms überschrieben wird und damit potentiell eine Sicherheitslücke entstehen kann.

Das ultimative Ziel eines jeden Angreifers auf ein System, ist der komfortable Zugang zu diesem und die Beschaffung möglichst vieler Zugriffsrechte. Auf dem Weg können

verschiedene Teilziele verfolgt werden. So wäre es mit Hilfe von Buffer-Overflows möglich eigenen Code im Arbeitsspeicher des Zielsystems auszuführen und darüber Keylogger oder Viren zu installieren, die weitere Zugriffe mit Hilfe neuer Funktionen ermöglichen könnten. Auch weitere Sicherheitslücken könnten durch die Manipulation anderer installierter Software geschaffen werden. Besonders interessant ist häufig der Zugriff auf dem Zielsystem gespeicherte Dateien.

Programme legen bei der Initialisierung und während der Laufzeit Daten und Anweisungen im Speicher ab. Dabei handelt es sich um Daten, die in der ausgeführten Software angezeigt oder vom Benutzer eingegeben werden. Speziell für erwartete Usereingaben muss vorher ein entsprechender Datenpuffer angelegt werden, indem die Eingabe dann gespeichert wird. Die Anweisungen werden genutzt um den Programmablauf zu modellieren. Dafür werden unter anderem Rücksprungadressen im Speicher abgelegt, die auf andere Speicherbereiche verweisen und damit den Kontrollfluss des Programms definieren. Wird so eine Rücksprungadresse durch Ausnutzung eines Buffer-Overflows gezielt überschrieben, kann ein Angreifer den Ablauf des Programms manipulieren, indem er die Rücksprungadresse auf eine andere Funktion oder Subroutine verweisen lässt. Außerdem wäre es möglich auf einen vorher durch die Usereingabe eingeschleusten Code zurück zuspringen. Dieses durchdachte Verfahren ist auch unter dem Stichwort (Shell-)Code-Injection bekannt.

1.3 Historische Angriffe

In der Vergangenheit wurden bekannte schwerwiegende Internetwürmer und Hacks häufig mit Hilfe von Buffer-Overflows ausgeführt und verbreitet. Aber auch die Umgehung von Sicherheitsvorkehrungen, z.b. eines Kopierschutzes oder die Entfernung von Software-Beschränkungen in Geräte-Firmware wurde mit Hilfe von Buffer-Overflows erreicht. Ein berühmtes Beispiel dafür sind die sog. „Jailbreaks“ von Apples Iphones. Sogar für kreative Projekte wurde diese Art der Programmmanipulation genutzt, so wurde auf dem 8-But Computer Game-Boy von Nintendo ein Spiel Namens Pokemon-Yellow von Innen heraus verändert. Durch Positionierung von Items im Inventar der Spielfigur wurde ein Bootstrap-Loader im Speicher des Game-Boys erstellt, der eine eigenständige Programmierung mit Hilfe von 8-Bit Shellcode ermöglichte.

Da aber weniger kreative Projekte und viel häufiger gefährliche, weit verbreitete Internet-Würmer ins Interesse der Medien und Gesellschaft raten, wird im Folgenden kurz auf den Blaster Worm (auch bekannt als Lovesan or MSBlast) eingegangen.

Als weiteres Interessantes Beispiel für gefährliche Sicherheitslücken ist in einem Autoradio aufgetreten, so war es möglich die Software des Fahrzeugcomputers zu manipulieren indem speziell manipulierte MP3 Dateien im Autoradio abgespielt wurden. Die Metadaten der MP3 Files hatten im System beim Auslesen einen Buffer-Overflow ausgelöst und somit Zugriff auf den Arbeitsspeicher gegeben, indem dann der MP3 angehängter

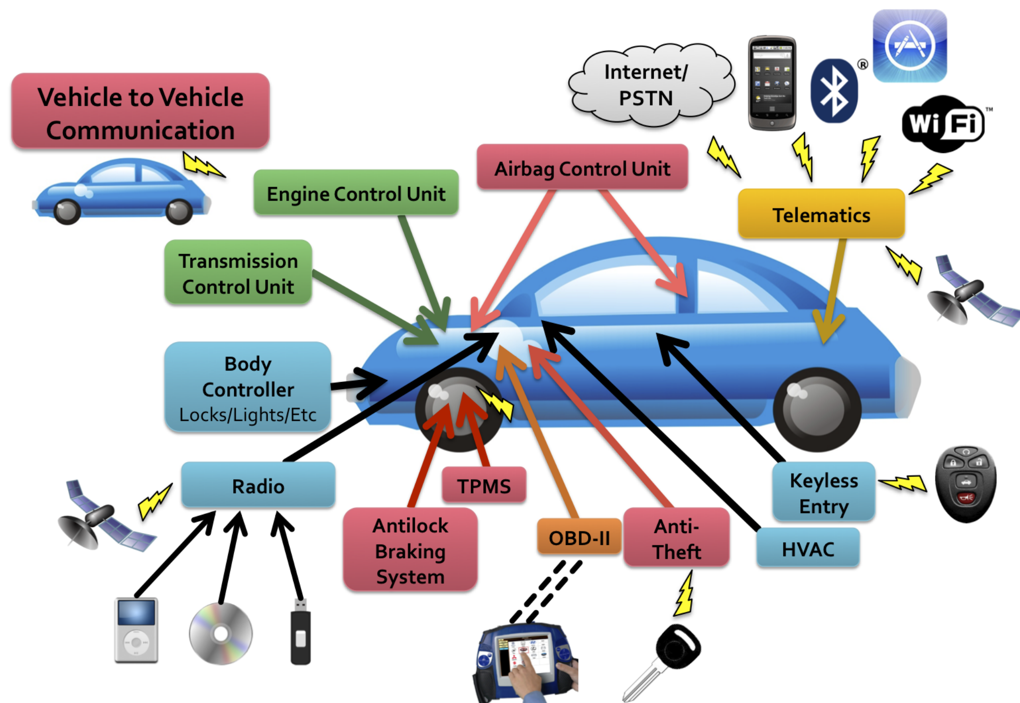
Shellcode ausgeführt wurde.

1.4 Speichermanipulation

Was passiert, wenn eine Benutzereingabe so lang ist, dass sie die, für einen Puffer reservierten, Bytes überschreitet? Tritt dieser Fall ein, versehentlich oder bewusst erzwungen, so kommt es zu einem Überlauf (Overflow) des Puffers. Dabei gehen die überzähligen Daten jedoch keinesfalls verloren, sondern werden in Speicherbereiche geschrieben, welche sich jenseits des eigentlichen Puffers befinden. Dies hat zur Folge, dass unter anderen sensitive Daten, welche zuvor in diesen Speicherbereichen abgelegt wurden, mit der überlangen Benutzereingabe überschrieben werden können. Zu diesen sensitiven Daten gehören beispielsweise Zeiger, aber auch wichtige Informationen, welche zur Steuerung des Programmflusses bzw. zur internen Speicherverwaltung dienen. Werden diese sensitiven Bereiche durch eine zufällig gewählte Benutzereingabe überschrieben, so kommt es in der Regel zu einer abrupten Beendigung des entsprechenden Prozesses. Wird der Inhalt der Benutzereingabe jedoch sehr genau konzipiert, so ist es möglich, die Daten der sensitiven Bereiche zur Programmsteuerung gezielt zu überschreiben, um dadurch den Programmfluss zu manipulieren, was wiederum dazu ausgenutzt werden kann, um beliebigen eingeschleusten Programmcode mit den Rechten des fehlerhaften Programms ausführen zu lassen [4]. Alle Buffer-Overflows sind das Produkt der schlecht entworfenen Programmen [1].

Das Problem betrifft besonders kompilierte Sprachen und auch solche Sprachen, die den direkten Zugriff auf Speicher und/oder Stack ermöglichen (z.B. C/C++), aber auch C# mit einem „unsafe“-Statement und Zeigern). Viele Programmierer machen sich oft keine Gedanken über die Überlaufe, da die Verwaltungskosten der Puffer Zeit und Konzentration kosten, also die Ressourcen, die man als kommerzieller Entwickler oft nur knapp zur Verfügung hat. Häufig ist die sichere Behandlung der Puffer nicht einfach zu realisieren. Die Entwicklung von „embedded devices“ ist dafür ein gutes Beispiel. Je nach dem Mikrocontroller muss eine hardwarenahe Sprache benutzt werden, die meist keine gute Abstraktionsklassen bietet. Werden Container und Algorithmen neu entwickelt, die in Hochsprachen bereits vorhanden sind, ist die Fehlerwahrscheinlichkeit sehr hoch.

Vor allem in der Automobilindustrie gibt es viele Beispiele, wie wenig Gedanken sich über die Sicherheit gemacht werden (vgl.[5]). 4 von 7 Angriffsvektoren, die erfolgreich ausgenutzt werden konnten, waren Buffer-Overflows.



Weiterhin werden wir detailliert erläutern, in welchen Fällen es zu solchen Buffer-Overflows kommen kann und wie diese mittels eines Exploits ausgenutzt werden können, um beispielsweise beliebigen externen Programmcode zur Ausführung zu bringen. Wir werden die Buffer-Overflows analysieren und schließlich mögliche Gegenmittel anschauen. Dafür erläutern wir die Softwarearchitektur auf hardwarenaher Ebene. Wir erklären außerdem, wie der Code aus der Sicht des Betriebssystem aussieht und nutzen dieses Wissen um die eigentliche Problematik zu erklären und eigene Beispiele zu implementieren.

1.5 Reverse Engineering

Um Pufferüberläufe zu finden und ihre Auswirkungen zu verstehen, müssen wir in der Lage sein die Software und den Einfluss der Software auf ihre Umgebung zu verstehen. Wir müssen also fähig sein, die Software zu zerlegen und potentielle Lücken zu finden. Hier kommt das Reverse Engineering ins Spiel. Reverse Engineering(RE) oder auch Rekonstruktion, bezeichnet den Vorgang, aus einem bestehenden, fertigen System oder einem meistens industriell gefertigten Produkt durch Untersuchung der Strukturen, Zustände und Verhaltensweisen, die Konstruktionselemente zu extrahieren. Um das nötige Wissen zu extrahieren, müssen wir die Softwareumgebung kennen.

1.6 Prozess und Speicherorganisation

Um das Prinzip von Angriffen, die auf dem Überlauf von Puffern beruhen, zu verstehen, ist es notwendig, sich gezielt mit der Prozess- und Speicherorganisation auseinan-

der zu setzen. Deshalb soll zunächst eine kurze Einführung in diesen komplexen Bereich erfolgen. Unter einer Binärdatei (Binary), bzw. einem Programm, versteht man generell eine ausführbare Datei, welche auf einem Datenträger gespeichert ist. Es gibt eine Reihe von verschiedenen Dateiformaten für solche ausführbaren Binärdateien. So wird beispielsweise auf Microsoft-Plattformen das Portable Executable Format (PE) eingesetzt. Ein weiteres Format für ausführbare Dateien stellt darüber hinaus das Executable and Linking Format, oder kurz ELF, dar, welches heutzutage von nahezu allen modernen UNIX-Varianten unterstützt wird [4]. Die binären Dateien, die auf der Festplatte oder sonstigen Speichermedien gespeichert sind, beinhalten Programme. Wird so eine ausführbare Binärdatei durch den Linker geladen und das Programm ausgeführt, so wird der entsprechende Programmcode in den Hauptspeicher geladen und dann von dem Zentralprozessor(CPU) ausgeführt. Das dann ablaufende Programm wird als Prozess bezeichnet. Wird dasselbe Programm gleichzeitig mehrmals gestartet, so handelt es sich dabei um mehrere verschiedene Prozesse, obwohl alle das gleiche Programm ausführen [3]. Das Programm läuft im Benutzermodus (user mode) und im Kern des Betriebssystems (kernel) und wird im privilegierten Modus ausgeführt. Dank des virtuellen Speichers, der die physikalischen Adressen und Register der Geräte auf virtuelle Adressbereiche überträgt, sieht der Speicher für alle Prozesse, so wie für das Betriebssystem selbst unterschiedlich aus. Wenn das Betriebssystem eine Bibliothek in den Speicher lädt, wird diese nicht wirklich nochmal in den Speicher geladen, sondern dessen physikalische Adresse in den virtuellen Speicher des Prozesses gemappt. Einige Bibliotheken und Programmcode sind jedoch oft an der gleichen Stelle für jeden Prozess, was ein Ausnutzen eines Fehlers im Programm ermöglicht.

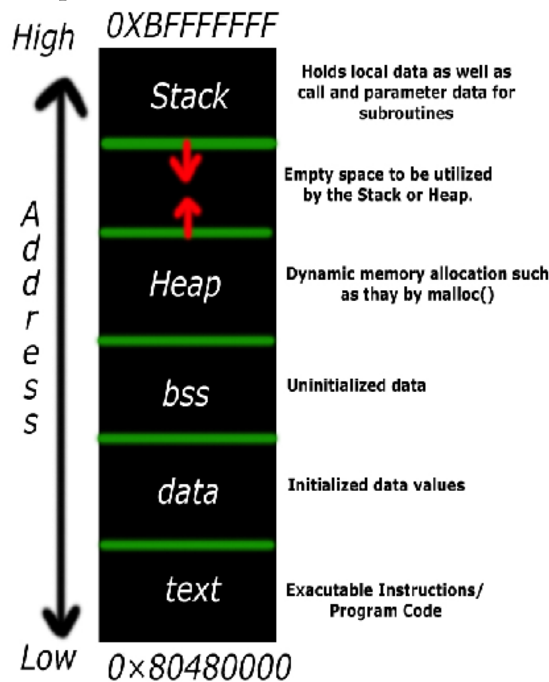
Die Binärdatei enthält neben dem Hauptprogramm noch verschiedene andere Anweisungen und Sektionen, die zur korrekten Funktion des Programms erforderlich sind. Wir werden das Konzept an einem ELF (executable and linkable format) File Format erklären und speziell den ausführbaren Typ davon als Beispiel nehmen. Ein ELF File besteht aus dem ELF Kopf, gefolgt vom Programm Kopf oder Sektionskopf oder beiden (vgl. [Linux Programmer's Manual ELF(5)]).

Das ELF File besteht aus

- Programm-Kopf Tabelle, die 0 oder mehrere Segmente beinhalten.
- Sections-Kopf Tabelle, die 0 oder mehrere Sektionen beschreibt
- Daten, die durch die Einträgen in beiden oberen Teilen referenziert werden.

Für mehr Details siehe [Portable Formats Specification, Version 1.1.]. Die verschiedenen Sektionen werden zu den verschiedenen Segmenten im ELF File gemappt. Das kann mit dem GNU binutils Befehl „`readelf -l datei`" nachvollzogen werden. Wenn das Programm aufgerufen wir, so werden die Sektionen zu den Segmenten im Prozess gemappt und die Segmente in Speicher geladen, wie es durch den ELF File beschrieben wird. Der Programmierer kann durch bestimmte Compiler spezifischen Einweisungen

die Variablen einer Sektion zuweisen, falls man eine Variable in einem bestimmten Speicherberich platzieren will.



Die für uns besonders interessanten Sektionen sind:

- Die **.bss** Sektion wird von mehreren Kompilern und Linkern als Teil des Data segments benutzt, die statisch allozierten Variablen beinhalten, die ausschließlich durch 0-Bits repräsentiert werden.
- Die **.data** und **.rodata** Sektionen beinhalten globale und statische Variablen, die von dem Programm explizit initialisiert werden. Die **.ro*** Variante ist nur lesbar. Z.B. in der Sprache C, wird die Zuweisung `char[] s = "hello"` in **.data** gespeichert, `const char[] s = "hello"` würde in **.rodata** abgelegt.
- Die **.text** Sektion beinhaltet die eigentlichen Assembleranweisungen des Programms. Um zu verhindern, dass der Prozess seine Instruktionen versehentlich modifiziert, kann auf diesen Bereich nur lesend zugegriffen werden (read-only). Jeder Versuch in diesen Bereich zu schreiben führt unweigerlich zu einem Fehler (Segmentation Violation oder Segfault) [4].

stackoverflow.com/questions/11884630/segments-within-a-executable-c-program

Wenn das Programm geladen wird, werden noch zwei für uns relevanten Segmente ins Leben gerufen:

- **Heap:** aus diesem Bereich wird Speicher alloziert (durch die malloc Funktion der Standardbibliothek oder durch Operator „new" in C/C++). Dieser Bereich fängt am Ende des **.bss** Segments an und wächst zu den höheren Adressen. Unter Unix gibt es 2 Systemcalls: **brk** und **sbrk** wodurch sich die obere Grenze des Heaps

verschieben lässt. Der Heap wird von allen vom Programm geladenen Modulen geteilt.

- Der **Stack:** ist eine Last-In-First-Out Datastruktur, in der die Rücksprungadressen, Parameter und je nach Compiler Optionen auch frame pointer gespeichert werden. In C/C++ werden hier lokale Variablen abgelegt und man kann sogar Code auf den Stack kopieren. Moderne Schutzmassenahmen gegen Buffer-Overflows würden jedoch das Ausführen von solchem Code verhindern.

Oft wird der so genannte 'frame pointer' verwendet um sich den Anfang des aktuellen lokalen Stackbereichs (frame) zu merken, sodass beim Aufruf einer beliebigen Funktion ihr frame pointer auf dem Stack gespeichert wird. Jedoch besteht die Möglichkeit dieses Verhalten auf einigen Architekturen durch ein Compilerflag auszuschalten. Der Stack fängt bei den hohen Adressen an und wächst nach unten. Deshalb können Zeichen, die nacheinander in den auf dem dem Stack allozierten Puffer geschrieben werden, die Daten der vorher aufgerufenen Funktionen überschreiben und sogar die Rücksprungadresse selbst.

Der Heap fängt hingegen bei den niederen Adressen an und wächst nach oben. Hier werden größere Speicherbereiche für Daten alloziert. Da sich hier normalerweise kein Code befindet (außer bei selbstmodifizierten Code und Just-In-Time Compilern), ist es vier schwieriger hierbei Fehler auszunutzen. In diesem Fall muss man mehr über die Speicheraufbau und das genaue Implementation des Heap Allokators wissen. Wenn ein Heap Bereich überläuft, können die nachfolgenden Blöcke modifiziet werden. Falls die Verwaltungsinformationen mitgespeichert werden, könnte man diese ebenfalls manipulieren und den Allokator dazu bringen beliebigen Code auszuführen.

1.7 Bezug zu C

In manchen Sprachen wird Speicher manuell verwaltet und in C/C++ kann Speicher wahlweise auf dem Stack oder dem Heap alloziert werden.. C wurde Anfang der 70'er Jahre bei AT&T von Dennis Ritchie entworfen. Der Sicherheitsaspekt war damals nicht so relevant wie heute. Das kann auch leicht an Beispielprogrammen gezeigt werden:

```
1 int main (void)
2 {
3   char c[10];       /* Allokation auf dem Stack. Haben wir ausreichend Platz? */
4                     /* Es gibt keine Möglichkeit dies im Voraus festzustellen. */
5   char* d = malloc(10); /* Allokation auf dem Heap */
6                     /* d ist NULL, falls nicht genug Speicher vorhanden */
7   return 0;         /* zurück zum Betriebssystem */
8 }
```

Da C/C++ sehr verbreitet sind (vgl. langpop.com), sind ihre Eigenheiten für die Buffer-Overflows besonders relevant. Eine dieser Eigenheiten sind die C-Strings.

1.7.1 C Strings

In C haben die Zeichenketten (Strings) ein bestimmtes Format das historisch besonders Platzsparend war. Die C-Zeichenkette besteht aus den beliebigen ASCII Zeichen die mit einem 0 Byte terminiert wird. Die Hilfsfunktionen, der im Nachhinein entworfenen C Bibliothek erwarteten keine Länge und haben das Ende der Zeichenkette nur am Nullzeichen erkannt. Jahre danach hat sich herausgestellt, dass dieses Verhalten ziemlich unsicher ist, da der Angreifer mittels der zu langen Eingabe das Nullzeichen überschreiben und so einen Fehler im Programm auslösen konnte [7]. Einige solcher unsicheren Funktionen werden mit neuen Standards ausgemustert. Zum Beispiel mit dem Eintritt der neuen C11 Standard wird die „gets“ Funktion, die eine Zeichenkette vom Standard Input liest und keine Eingabelänge überprüft ausgemustert. Die unsicheren Funktionen sind einer der wichtigsten Quellen für die Stack-basierte Puffer Überläufe. C++ bietet hierfür eine Alternative zu C-Strings unter dem Namen `std string`, wo die Länge des Strings mitgeführt wird.

1.7.2 Aufrufkonventionen

Aufrufkonventionen sind Methoden, mit der in Computerprogrammen einer Funktion Daten übergeben werden. Sie sind Architektur-, Compiler- und Programmiersprachenspezifisch. Für C sind diese in dem sog. ABI(Application Binary Interface) definiert. Eine Aufrufkonvention kann vom C/C++ Entwickler für jede Funktion einzeln gewählt werden. Für uns relevant ist „cdecl“ (wird von GNU C Compiler verwendet). Für unseres Beispiel haben wir GCC benutzt, also die cdecl Aufrufkonvention. Bei der cdecl Aufrufkonvention werden die Argumente der Funktion auf dem Stack in verkehrter Reihenfolge übergeben. Der Aufrufer räumt den Stack auf und falls `eax`, `ecx` und `edx` Register vom Aufrufer benutzt wurden, so werden diese von ihm auf dem Stack gespeichert.

1.8 Typen von Buffer-Overflows

Buffer-Overflows lassen sich generell in 3 Typen aufteilen [1]: Stack Überlaufe, Heap Überlaufe und die Überläufe, die durch `printf(3)` Styleformatierung entstehen.

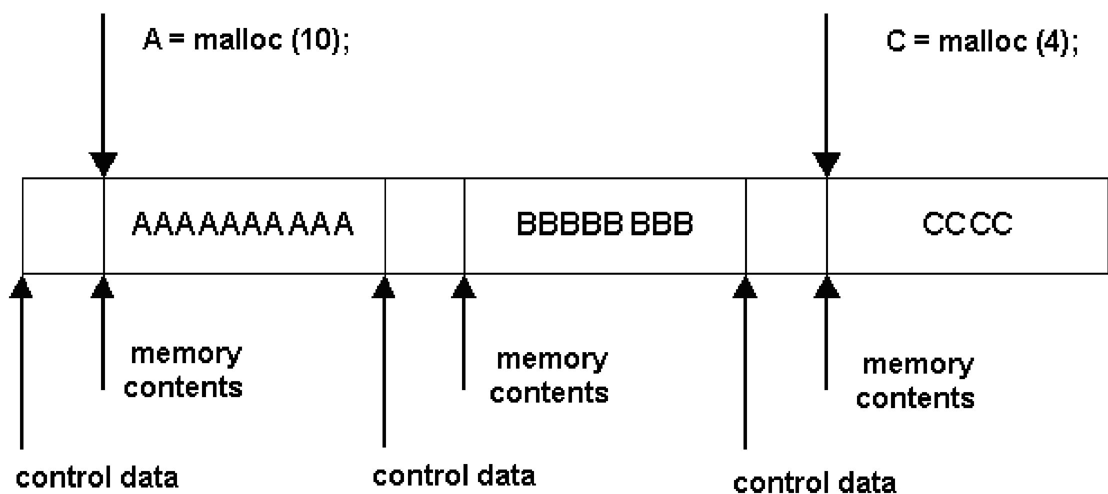
1.8.1 Stack Überlauf

In der Software geschieht ein Stacküberlauf, wenn ein Programm etwas auf eine Adresse auf dem Stack schreibt, die sich außerhalb der beabsichtigten Datenstruktur befindet. In der Regel ist diese Struktur ein Puffer mit einer festen Länge [2]. Diese Art von Angriff benutzt die Tatsache, dass die CPU bei Aufruf einer Funktion, die Adresse des nächsten

Befehls auf dem Stack speichert. Hinzu kommt der Fakt, dass der Compiler den 'frame pointer' benutzt der ebenfalls auf dem Stack abgelegt wird. Falls ein Puffer auf dem Stack alloziert wird, in den die Benutzereingabe ungeprüft reinkopiert wird, so kann der Angreifer diese Eingabe so überdimensionieren, dass die dahinterliegenden Daten (z.B. der alte frame-pointer oder die Adresse des nächsten Befehls) überschreiben werden. Nicht alle Programmiersprachen erlauben dem Entwickler auf diese Weise auf den Stack zuzugreifen, wie in C/C++ möglicht ist. Bei Betrachtung der 10 populärsten Sprachen (langpop.com), sind nur C und C++ in der Lage Datenstrukturen auf dem Stack explizit zu allozieren. Aus diesem Grund kann dieser Typ des Überlaufs nicht allgemein und unabhängig von C/C++, ihren Eigenheiten und Kompilern betrachten.

1.8.2 Heap

Im Unterschied zum Stack-Overflow haben die Heap-Overflows viele verschiedene Techniken und Konsequenzen. Heap-Speicher unterscheiden sich zu Stack-Speichern insofern, dass er zwischen den Funktionsaufrufen persistent ist. Das bedeutet, dass Speicher, die durch `malloc/new` alloziert wurden solange alloziert bleiben, bis `free/delete` aufgerufen wurden (vgl. [1]). Das bedeutet, dass der Überlauf nicht bemerkt wird, solange der Speicherbereich später nicht benutzt wird. Die Struktur des Heaps ist implementierungsspezifisch, jedoch funktionieren die meisten so, dass sie mehrere Sammlungen (Pools) von Blöcken gleicher Größe verwalten (vgl. [1]). Dabei hat jeder Pool eine unterschiedlich Größe, so dass die Allokationen, die am besten in den nächst-größeren Pool passen, einen Block daraus reservieren. Die Verwaltungsinformationen des Blocks werden üblicherweise am Anfang des Blocks gespeichert und die Blocks oft nacheinander alloziert. Durch gezieltes Überlaufen eines Blocks könnte man die Informationen des nächsten Blocks überschreiben.



Das Ziel des Heapüberlaufs ist einen Funktionspointer zu überschreiben. Eine Möglichkeit ist einen virtuellen Tabellenzeiger einer virtuellen C++ Klasse zu überschreiben und so einige Methoden zu überladen.

Formatierungsfunktionen

Diese Schwachstelle existiert wegen einiger Eigenschaften von C. C erlaubt eine variable Anzahl von Argumenten in der Stringformatierungsfunktion. Da die Anzahl der Parameter zur Kompilierzeit nicht bekannt ist, muss sich eine Funktion auf den Formatierungsstring verlassen, der allerdings vom Angreifer ausgenutzt werden kann. Eine weitere Schwachstelle ist, dass der Angreifer einen zu langen String-Parameter übergeben könnte und die Formatierungsoperation beim Einlesen den Zielpuffer überlaufen lässt. Dieses Problem kann umgangen werden, indem die Eingabe des Nutzers strenger validiert wird. In der Realität passiert das leider oft nicht oder wird als unwichtig eingestuft. Und so kann der Angreifer Schadcode an einer definierten Stelle in RAM injezieren, und wenn sich der Zielpuffer auf dem Stack befindet, direkt durch Modifikation der Rücksprungadresse die Kontrolle über das Programm übernehmen.

1.9 Shellcode

Der Code, der mit Hilfe eines Fehlers (auch als Exploit bekannt) eingeschleust wird, ist unter dem Namen „Shellcode“ bekannt. Der häufigste Zweck eines solchen Codes ist einen Kommandointerpreter bzw. Shell zu starten und darüber dem Angreifer kompletten Zugriff auf den Rechner zu erlauben. Es gibt viele Beispiele dafür im Internet. Shellcode hat besondere Anforderungen: er muss klein sein, da man oft nicht viel Platz für den Code hat und er muss auf der niedrigsten Ebene arbeiten, um möglichst viele Modifikations- und Kontrollmöglichkeiten über das Betriebssystem zu bekommen. Kein Wunder, dass sicht fast alle Angreifer für Assembler entscheiden. Wir beschränken uns auf die x86 Assemblersprache, da die Architektur auf fast jedem PC vertreten ist und deshalb das Testen sehr vereinfacht.

1.9.1 Null byte Problem

In dem Fall, dass ein C String für den Pufferüberlauf benutzt wurde, muss aufgepasst werden, dass keine Anweisung ein Null Byte enthält (0x00), damit die Stringmanipulationsfunktionen der C Bibliothek immer den gesamten Shellcode kopieren und nicht nach dem Null Byte abbricht. Für diesen Zweck gibt es spezielle Programme, die Befehle mit Nullbytes in Befehle ohne Nullbytes umwandeln. Alternativ kann der Shellcode selbst modifiziert werden, sodass andere Maschinenbefehle der x86 CISC Architektur, ohne Nullbyte, verwendet werden. Wenn zum Beispiel eine 0 direkt im Register geladen wird (`mov eax, 0`), dann enthält dieser Befehl ein Nullbyte, das als unmittelbarer Wert im Befehl vorkommt. Die Alternative dazu wäre ein Register mit sich selbst exklusiv zu „verodern“ (`xor eax, eax`).

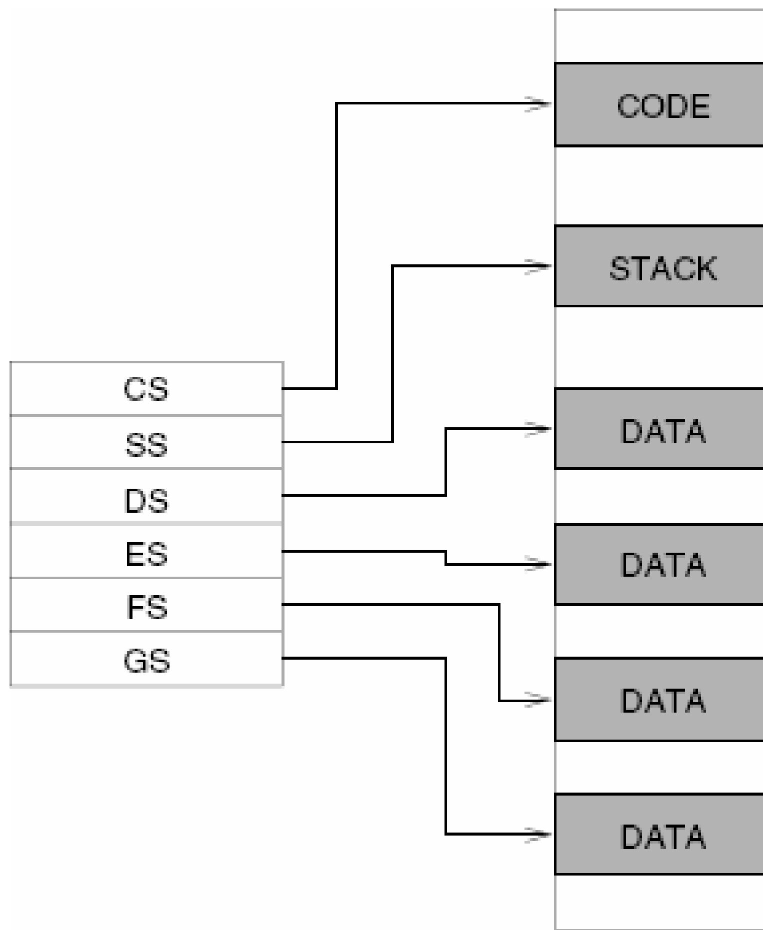
1.10 x86 Architektur

Als kleine Einführung in den Assembler, schauen wir uns zuerst die x86 Architektur an. Für eine detaillierte Beschreibung können die x86 Manuals[1] zu Rate gezogen werden.

X86 hat einen CISC(Complex instruction set computing) Befehlssatz. Diese Architektur hat 7 Allzweckregister und ein Stack Pointer Register (esp), der auf die letzte gespeicherte Zelle zeigt. Darüber hinaus gibt es ein Programm Zähler (eip) und einige Segmentregister. X86 ist im Allgemeinen eine 2 Operand Registermaschine, wobei das Zielregister eines der Operanden ist. Die x86 Familie hat für die Computerarchitekturen beachtliches Alter und das muss man im Kopf behalten, wenn man den Maschinencode anschaut. Moderne Implementationen sind immer noch zu dem ursprünglichen 8086 kompatibel. Viele Befehle werden nicht mehr vom Compiler genutzt um Code zu erzeugen, viele sind unnötig komplex. Der Vorteil ist jedoch der Fakt, dass viele oft benutzte Befehle nur ein Byte lang sind und im Allgemeinen die Codegröße und somit die benötigte durchschnittliche Speicherbandbreite/Befehl minimal ist [codeden]. Mit der Zeit wurden die Register von 16 auf 32 Bits erweitert (diese Register haben ein e- Präfix), und später auf 64 bit (r-Präfix) mit mehreren Registern und Standard Vektoreinheit.

Darüber hinaus existieren noch die Segmentregister, die in der 16-Bit Era Speicher in 2^{16} Bit Blöcke unterteilt haben und somit den Adressraum um den Faktor 16 (insgesamt auf ein Megabyte) vergrößert haben. Mit Intel 80286 gab es den ersten halb fertigen Protected Mode, der mit dem Intel 80386 richtig implementiert wurde. Für uns sind die Segmentregister nur soweit relevant, als dass sie als Voreinstellungssegment für jeweils Stack Pointer, Programmzähler und das Datensegment implizit benutzt werden. Zum Beispiel: Die effektive Adresse der letzten Stackzelle ist `ss:esp`, und die effektive Adresse des nächsten Befehls ist `cs:eip` was im Endeffekt so viel wie die Adresse `cs * 16 + eip` bedeutet.

[1] intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html



X86 benutzt Little Endian. Das bedeutet, dass die mehrbytigen Integer im Speicher byteweise in der umgekehrten Reihenfolge gespeichert werden. Der Vorteil von Little Endian ist die Tatsache, dass der Zeiger nicht abhängig von dem Typ modifiziert werden muss. Wenn wir zum Beispiel 16 gespeichert haben, so werden wir mit dem gleichen Zeiger, egal wie die Interpretation aussieht (z.B. ein Zeiger auf ein Byte oder auf ein 32 Bit Integer), immer 16 auslesen. Bei dem Big Endian muss der Wert des Zeigers je nach dem Typ angepasst werden. Der Nachteil ist, dass die Integer Werte im Speicher in umgekehrter Reihenfolge stehen und man beim Reverse Engineering ständig umrechnen muss.

1.11 Assembler

Der Code in der **.text** Sektion des Programms ist in der Maschinensprache (CPU Befehlssatz) geschrieben. Die Assemblersprache ist eine spezielle Programmiersprache, welche die Maschinensprache einer spezifischen Prozessorarchitektur in einer für den Menschen lesbaren Form repräsentiert. Jede Computerarchitektur hat folglich ihre eigene Assemblersprache.

Die Assembler Befehle bestehen aus einem Opcode (Maschinenbefehl ohne Operanden) und ein oder mehreren Operanden. Wir benutzen die Intel-Notation, in der sich das

Zielregister auf der ersten und das Quellregister auf den zweiten Stelle befindet. Der populärste Maschinenbefehl ist „`mov`“, der ein Register- oder Speicherinhalt in ein anderes Register oder Speicherzelle kopiert. Im Befehlssatz von x86 gibt es auch logische und arithmetische Operationen, bedingte und unbedingte Sprünge so wie dedizierte Operationen zur Stackmanipulation(`push, pop`). Außerdem können Softwareinterrupts mittels der int Instruktion erzeugt werden. Während des Softwareinterrupts, wechselt das Betriebssystem in den Kernel Modus und führt den Systemaufruf (system call) aus. Um einen Wert des Register als Adresse zu interpretieren und darauf zuzugreifen werden die eckigen Klammern benutzt. `mov eax, 1` bedeutet „kopiere 1 in das Register `eax`“. `mov eax, [1]` bedeutet so viel wie „kopiere den Inhalt der Zelle auf der Adresse 1 ins Register `eax`“. Die einzige Ausnahme ist der `lea` Befehl. X86 hat keine „Multiply-Accumulate“ Instruktion, kann jedoch den `lea` Befehl dazu missbrauchen. Lea steht für „load effective address“, lädt aber nichts aus dem Speicher. `lea z, [a + i * b +j]` bedeutet `z = a + i * b + j`, wobei `i` und `j` unmittelbare Werte sind, die kleinere Potenzen von 2 sind. Der Hintergrund dazu ist die Methode wie die Arrays der Datenstrukturen adressiert werden. Hier sei `a` der Anfang des Arrays, `b` der Index eines Elements und `j` die Verschiebung eines Elements vom Anfang der Datenstruktur. Damit es funktioniert, muss die Strukturgröße `i` auch Potenz von 2 sein. Um die Daten in Assembler zu deklarieren, benutzt man die d-Direktive. `db` steht für Bytes, `dw` für 16-Bit Worte (x86 war ursprünglich 16 Bit), `dq` für 32 Bit Integer (Quad Byte). Der Funktionsaufruf geschieht mit dem „`call`“ Befehl, der die Adresse des nächsten Befehls auf den Stack legt und die CPU auf eine Zieladresse springen lässt. Die dafür inverse Operation ist „`ret`“. Sie holt die auf dem Stack liegende Adresse in den Programmzähler `eip`. Das Programm wird ab dieser Adresse weiter ausgeführt.

1.12 Shellcode Beispiel

Shellcode für Windows und für Unix-basierte Betriebssysteme ist konzeptuell sehr unterschiedlich. Beim Entwurf von Shellcode für Windows, muss beachtet werden, dass ungleich den verschiedenen Unix Varianten, der Weg für einige Aufgaben nicht so einfach ist. Ein Betriebssystemaufruf über einen Software-Interrupt (system call) ist ein gutes Beispiel für diesen Unterschied. Windows hat zwar auch system calls, jedoch diese sind von der jeweiligen Version bzw. den spezifischen Service Packs abhängig. In Windows wird auf die Betriebssystem Funktionalität mit Hilfe von Bibliotheken zugegriffen. Das erfordert das Wissen, wo im Speicher die Bibliotheken vorhanden sind und ob diese überhaupt in einen lokalen Adressraum gemappt sind. In Unix hingegen, kann ein Betriebssystemaufruf einfach mit `execve(3)` ausgeführt werden. Die Liste von allen Linux Syscalls findet sich in `<linux_source>/include/asm-x86/unistd_64.h`.

[skullsecurity.org/wiki/index.php/Example_4] Hier ein Shellcode Beispiel unter x86 Linux. Dieses Programm führt den Kommandointerpreter „`sh`„ aus, der sich standard-

mäßig in /bin/ befindet.

```
BITS 32   ; Direktive für 32 Bit Modus
jmp short two  ; springe nach unten für die Adresse von "/bin/sh"
one:
  pop ebx        ; Hole die Rücksprungadresse vom Stack
                   ; und speichere es ins ebx
  ;execve(const char *filename, char *const argv [], char *const envp[])
  xor eax, eax   ; eax = 0
  mov [ebx+7], al  ; terminiere den "/bin/sh" String mit 0
  mov [ebx+8], ebx ; speichere die Adresse von /bin/sh hinter dem String
  mov [ebx+12], eax ; und speichere 0 hier

  mov al, 11     ; execve syscall hat Nummer 11
  lea ecx, [ebx+8] ; Adresse des ersten Parameters (/bin/sh) in ecx
  lea edx, [ebx+12] ; Adresse des zweiten Parameter, hier liegt die 0
  int 0x80       ; Software Interrupt: führe den Systemaufruf aus.
two:
  call one       ; Benutze den call um die Adresse von /bin/sh auf dem
               ; Stack abzulegen.
  db '/bin/sh'   ; erstes execve Parameter.
```

jmp short ist ein relativer Sprungbefehl. Das ist wichtig, weil zur Laufzeit die genaue Adresse des Shellcodes noch unbekannt ist.

1.13 Sicherheitsmaßnahmen

1.13.1 Canaries

Bei sog. "Canaries„ handelt es sich um bekannte Werte, die zwischen Puffer und Kontrolldaten (control data) auf den Stack geschrieben werden, um Buffer-Overflows zu bemerken. Das Prinzip ist dabei, dass bei einem Buffer-Overflow als erstes die Canarie überschrieben würde und dass das Betriebsystem während der Laufzeit überprüft, dass die Canarie vorhanden und unverfälscht ist. Falls nicht, würde ein Fehler ausgegeben, der dann behandelt werden kann, oder zur Beendigung des Programms führt.

Das Wort "Canarie„ (zu Deutsch: Kanarienvogel) ist eine Anspielung auf den Umstand, dass Kanarienvögel früher in Kohleminen benutzt wurden, um frühzeitig den Austritt von giftigen Gasen festzustellen, da sie auf Grund ihrer kleinen Lungen schneller als Menschen von den Giftgasen betroffen sind.

In der Informatik werden drei Arten von Canaries benutzt: Terminal, Random, und Random XOR.

Terminator Canaries

Da die meisten Buffer-Overflow Angriffe auf String Operationen basieren, die bei bestimmten Zeichen terminiert werden. Deshalb werden Terminator Canaries aus NULL-Terminatoren, CR, LF und -1 erstellt. Ungünstigerweise sind diese Canaries bekannt und können deshalb von einem Angreifer mit dem eigenen Wert überschrieben werden, sodass die Betriebssystemskontrolle das eigentlich überschriebene Canarie als gültig erkennt und weiterhin Buffer-Overflows zulässt.

Random canaries

Random Canaries setzen an der Schwachstelle der Terminator Canaries an und werden zufällig generiert um zu verhindern, dass der Angreifer den Wert der Canarie im Voraus kennt. Normalerweise ist es nicht möglich, den Wert der Canarie während der Laufzeit auszulesen, außer vom Betriebsystem um Buffer-Overflows zu erkennen. Das wird erreicht, indem eine Random Canarie bei der Programm Initialisierung generiert wird und in einer globalen Variable gespeichert wird. Diese Variable ist üblicherweise von nicht gemappten Seiten (unmapped pages) umgeben, sodass ein direkter Leseversuch einen Segmentation Fault erzeugt und das Programm terminieren lässt. Dennoch kann es möglich sein die Canarie auszulesen, wenn der Angreifer weiß, wo sie sich befindet, oder das Programm dazu ausnutzen kann vom Stack zu lesen.

Random XOR canaries

Random XOR Canaries sind zufällig generierte Canaries, die mit dem Teil der Kontrolldaten (control data) geXORt werden. In diesem Fall wird der Wert der Canarie falsch, sobald sie selbst oder die Kontrolldaten verändert werden. Genauso wie bei der Random Canarie, kann die Canarie zur Laufzeit vom Stack gelesen werden, jedoch ist es schwieriger für den Angreifer eine neue Canarie so zu erstellen, dass sie mit dem gleichen Algorithmus erzeugt wird und zu dem veränderten Kontrolldaten passt. Random Canaries schützen zwar die Kontrolldaten, aber keine anderen Daten und auch nicht die Funkionspointer selbst. Wenn ein String so überläuft, dass er Funktionspointer überschreibt, kann dieser direkt auf Shellcode zeigen und diesen aufrufen lassen.

1.13.2 Address Space Layout Randomization

Bei der Address Space Layout Randomization (ASLR) handelt es sich um ein Sicherheitsfeature gegen Buffer-Overflows. ALSR erschwert einige Arten von Angriffen, indem es dem Angreifer Schwierigkeiten beim finden von Zieladressen im Speicher bereitet. So müssen Angreifer, die einen "return-to-libc„ Angriff ausführen möchen, zuerst die richtige Adresse im Speicher finden, während andere Angreifer versuchen ihren eigenen Shellcode in den Stack zu injizieren und auszuführen, müssen auch diese zuerst den richtigen

Stack finden. In beiden Fällen hält das Betriebssystem mit ALSR die relevanten Speicheradressen verschleiert vor den Angreifern. Die Adressen müssen also erraten werden, wobei eine falsch geratene Adresse höchstwahrscheinlich zum Absturz des Programms führt und dementsprechend nur ein Versuch existiert.

2 Aufgaben

2.1 Warm werden mit Hexadezimalzahlen

Hexadezimal wird zur komfortableren Verwaltung des Binärsystems verwendet. Im Gegensatz zum Dezimalsystem eignet sich das Hexadezimalsystem mit seiner Basis als vierte Zweierpotenz ($16 = 2^4$) zur einfacheren Notation der Binärzahlen, da stets eine feste Anzahl Zeichen zur Wiedergabe des Datenwortes benötigt wird. (So kann die binäre 1111 einfach als einstelliges F statt als zweistellige 15 dargestellt werden).

```
Hexadezimal (Zahlensystem zur Basis 16)
0    1    2    3    4    5    6    7    8    9    A    B    C    D    E
Dezimal (Zahlensystem zur Basis 10)
0    1    2    3    4    5    6    7    8    9   10   11   12   13   14
```

1. Stelle die dezimale Zahl 10 im hexadezimalen Zahlensystem dar.
2. Stelle die dezimale Zahl 16 im hexadezimalen Zahlensystem dar.
3. Welche hexadezimale Zahl folgt auf F?
4. Welcher dezimalen Zahl entspricht der hexadezimale Wert FF?
5. Stelle die dezimale Zahl 164 im hexadezimalen Zahlensystem dar.
6. Welcher dezimalen Zahl entspricht der hexadezimale Wert 1000?
7. If someone told you, he could see DEAD people, how many people does he see?

2.2 Hauptaufgaben

In den folgenden Aufgaben gilt es, die Ausgabe bzw. das Verhalten des Programms *example1.exe* allein durch die Eingabe von bestimmten Zeichenketten zu beeinflussen, ohne das Programm selbst zu verändern. Dazu muss der Ablauf des Assembler-Programms untersucht werden und Buffer-Overflow als Angriff angewendet werden.

Bei *example1.exe* handelt es sich um ein kurzes Programm, das in C geschrieben und mit Mingw32 GCC kompiliert wurde. Den ursprünglichen C-Code findest Du im Abschnitt „Eclipse“ auf diesem Aufgabenblatt.

Zum Ausführen des Programms ist es sinnvoll die Bash-Konsole zu benutzen. Den Link zum Aufruf der Konsole findest auf dem Desktop der VM. Beim Aufrufen des Links landest Du automatisch im richtigen Verzeichnis

```
C:\Dokumente und Einstellungen\svsadmin\Desktop\workspace\example1\Debug
```

und kannst das Prgramm wie folgt aufrufen:

```
./example.exe
```

Das Programm fragt den Benutzer nach Eingabe eines Namens als Zeichenkette. Nach der Eingabe gibt das Programm eine Ausgabe aus und wird beendet.

Für die Bearbeitung der Hauptaufgaben benutzt Du zum Disassemblieren das Programm *IDA Demo*. Öffne dazu den auf dem Desktop abgelegten Link *IDA Demo*. Im Dialogfenster *Quick Start* drücke auf *New*. Dann öffne im Dialogfenster *Select file to disassemble* die Binärdatei *example1.exe*. Im nächsten Dialogfenster *Load a new file* drücke einfach auf *OK*.

Mache Dich erst ein wenig vertraut mit der IDA-Umgebung. Zum Debuggen des Programms wähle erst in dem oberen Listenfeld, in dem nach dem Start von IDA erst *No debugger* ausgewählt ist, den Eintrag *Local Win32 debugger* aus. Danach kann das Programm gestartet werden durch Drücken auf den Shortcut $F9$ (weitere nützliche Shortcuts findest Du im Abschnitt „Interactive Disassembler (IDA)“ auf diesem Aufgabenblatt).

Zur Bearbeitung der folgenden Aufgaben ist es sinnvoll sich genauer anzuschauen, was auf dem Stack geschieht. Dazu kannst Du gerne die Tabelle 2.1 zum Notizen-Machen benutzen.

1. Finde eine Eingabe, so dass der Inhalt der C-Variablen `var1` und `var2` ausgegeben wird.

2. Finde eine Eingabe, so dass „[Your name] is a not so good hacker!“ ausgegeben wird.

3. Finde eine Eingabe, so dass „[Your name] is a good hacker!“ ausgegeben wird.

4. Finde eine Eingabe, so dass der Aufruf der `main`-Methode neugestartet wird und der Benutzer erneut zur Eingabe seines Namens aufgefordert wird.

2.3 Zusatzaufgaben

Baue in dem Programm ein Cannary-Schutz ein. Dazu musst Du das Programm mit dem Compiler-Flag `-fstack-protector-all` neukompilieren. Benutze hierfür Eclipse:

Öffne das Projekt *example1* in Eclipse. Gehe dann durch Rechtsklick auf das Projekt-Symbol im *Project Explorer* in den *Properties*-Dialogfenster. In diesem gehe unter *C/C++ Build* → *Settings* im Tab *Tool Settings* zu *GCC C Compiler* → *Miscellaneous* und trage im Textfeld *Other flags* zusätzlich den neuen Flag ein. Danach trägst Du den neuen Flag auch unter *MinGM C Linker* → *Miscellaneous* im Textfeld *Linker flags* ein. Nach dem Beenden des *Properties*-Dialogfensters mit *OK*, musst Du das Programm neukompilieren. Dazu kannst Du den Shortcut $Ctrl + B$ benutzen.

Nun kannst Du die kompilierte Datei zum Untersuchen wieder in IDA aufmachen.

Stack				Adresse	Kommentar
				10	
				14	
				18	
				$1C$	
				20	
				24	
				28	
				$2C$	
				30	
				34	
				38	
				$3C$	
				40	
				44	
				48	
				$4C$	
				50	
				54	
				58	
				$5C$	

Tabelle 2.1: Darstellung des Stacks für Notizen.

1. Funktionieren noch die Buffer-Overflow-Angriffe aus den Hauptaufgaben?
2. Was wurde im Assembler-Code verändert?
3. Auf welche Speicherstelle wurde das Cannary gesetzt und mit welchem Wert initialisiert?
4. Ist es dennoch möglich diesen Schutz irgendwie zu umgehen, um einige der in den Hauptaufgaben gesetzten Angriffe zu erzielen?

2.4 Hinweise und Hilfestellungen

In den folgenden Unterabschnitten findest Du einige Hinweise und Hilfestellungen, die bei der Bearbeitung der Aufgaben hilfreich sein könnten.

2.4.1 Eclipse

Das Eclipse-Project ist zu finden unter

```
C:\Dokumente und Einstellungen\svsadmin\Desktop\workspace\example1
```

In der Datei *example1.c* befindet sich das für die Bearbeitung der Aufgaben benutzte C-Programm, das vollständig in Listing 2.1 aufgelistet ist. Im Code sind Kommentare als Hilfestellungen zum Verständnis und als Hinweise für den Lösungsweg angegeben.

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
  const char* pNotSoGood = "not so good"; /* Zeiger auf eine Zeichenkette. */
  int32_t var2 = -256; /* Lokale Variable auf dem Stack. */
  int32_t var1 = -1; /* Lokale Variable auf dem Stack. */
  char vName[10] = "123456789"; /* Array aus 10 Zeichen auf dem Stack. */

  printf("Enter your name (max. 9 characters): \n");
  /* Nutzer-Eingabe von der Kommandozeile wird in Puffer vName geschrieben.
   * Die Laenge beim Schreiben in den von vName adressierten Speicher wird
   * hierbei nicht ueberprueft! Als letztes Zeichen wird '\0' geschrieben. */
  gets(vName);

  if (var1 == 0x61626261) /* 61h 62h 62h 61h <-> a b b a */
    /* Ausgabe der lokalen Variablen var1 und var2. Die Ausdruecke %08xh
     * werden jeweils durch 8-stellige Hex-Werte der jeweiligen Variable
     * ersetzt. */
    printf("var1=%08xh\nvar2=%08xh\n", var1, var2);

  if (var2 != 0x6e65706f) /* Welche Zeichen werden hier repraesentiert? */
    /* Der Ausdruck %s wird durch die von vName adressierten Zeichenkette
     * ersetzt. Das Auslesen der Zeichenkette terminiert beim Zeichen '\0'. */
    printf("Your name is: %s\n", vName);
  else
    /* Hier wird die maximale Laenge bei der Ausgabe der beiden
     * Zeichenketten einmal auf 10 und einmal auf 20 gesetzt.
     * Welche Adresse befindet sich im Zeiger pNotSoGood? */
    printf("%.10s is a %.20s hacker!", vName, pNotSoGood);

  /* Der Aufruf von getchar() fuehrt dazu, dass das Komandofenster nicht
   * sofort schlieszt, sondern erst beim Druecken der Enter-Taste, so dass die
   * Ausgaben noch sichtbar sind. */
  getchar();
  return 0;
}
```

Listing 2.1: Das Ausgangsprogramm in C.

2.4.2 Interactive Disassembler (IDA)

IDA ist ein Programm zur interaktiven Disassemblierung von ausführbaren Binärdateien.

Zum Bearbeiten der Aufgaben in IDA ist die Benutzung einiger Shortcuts sehr hilfreich. In Tabelle 2.2 sind nützliche Shortcuts zum Arbeiten im IDA-Editor und in Tabelle 2.3 zum Debuggen aufgeführt. Die vollständige Liste der Shortcuts ist durch *Options* → *Shortcuts* aufrufbar.

Shortcut	Beschreibung
$space$	Ansicht Text/Graph wechseln
$alt + t$	Textsuche
n	Variable umbenennen
;	Kommentar einfügen
Esc	Springe zur vorherigen Cursorposition
$Ctrl + Enter$	Springe zur nächsten Cursorposition

Tabelle 2.2: Nützliche IDA-Shortcuts zum Arbeiten im IDA-Editor.

Shortcut	Beschreibung
$F2$	Breakpoint setzen
$F4$	Führe aus bis zum Cursor
$F7$	Springe in den Schritt rein
$F8$	Nächsten Schritt ausführen
$Ctrl + F7$	Führe aus bis zum Return

Tabelle 2.3: Nützliche IDA-Shortcuts zum Debuggen.

2.4.3 Assembler

In der Aufgabe wird eine 32-bit Maschine mit einem $x86$-Assembler-Befehlssatz verwendet. In diesem Abschnitt werden die für die Bearbeitung der Aufgaben relevanten Befehle und Register kurz erläutert.

Ein Maschinenbefehl besteht aus dem Opcode (Operation Code) gegebenenfalls gefolgt von bis zu zwei Operanden. Jedem Opcode wird in der Assemblersprache jeweils ein Mnemonic zugeordnet, ein kurzes Befehlswort. Ein Opcode zusammen mit den zugehörigen Operanden bildet einen Maschienenbefehl.

Ein Register ist eine Speicherstelle auf dem Prozessor. In unserem Fall ist jedes Register 32 bit groß. In Tabelle 2.4 sind die für die Bearbeitung der Aufgaben relevanten Register aufgeführt.

Die Operanden können aus Registern, Speicherstellen durch Angabe von Adressen oder unmittelbar durch Konstanten entnommen werden. So existieren unterschiedliche Adressierungsarten. Einige davor sollen hier anhand des `mov`-Befehls kurz erläutert werden:

Name	Bemerkung
`eax`	allgemein verwendbar
`ebx`	allgemein verwendbar
`ecx`	allgemein verwendbar
`edx`	allgemein verwendbar
`esi`	Quelle für Stringoperationen
`edi`	Ziel für Stringoperationen
`ebp`	Basepointer
`esp`	Stackpointer
`eip`	Instructionpointer
`efl`	Flags

Tabelle 2.4: Relevante Register.

Registeradressierung: Der Wert eines Registers wird in ein anderes übertragen.

```
mov ebx, edi
```

Unmittelbare Adressierung: Die Konstante wird in das Register übertragen.

```
mov ebx, 1000
```

Direkte Adressierung: Der Wert, der an der angegebenen Speicherstelle steht, wird in das Register übertragen.

```
mov ebx, [1000]
```

Register-indirekte Adressierung: Der Wert, der an der Speicherstelle steht, die durch das zweite Register bezeichnet wird, wird in das erste Register übertragen.

```
mov ebx, [eax]
```

Basis-Register-Adressierung: Der Wert, der an der Speicherstelle steht, die sich durch die Summe des Inhalts des zweiten Registers und der Konstanten ergibt, wird in das erste Register übertragen.

```
mov eax, [10+esi]
```

In der folgenden Liste werden die für die Bearbeitung der Aufgaben benötigten Befehle kurz erläutert:

push Quelle Speichert den angegebenen Operanden auf dem Stack ab und erniedrigt den Stackpointer (`esp`-Register) entsprechend.

pop Ziel Lädt das oberste Wort vom Stack in das Ziel und erhöht den Stackpointer (`esp`-Register) entsprechend.

lea Ziel, Quelle Lädt die Adresse des Quelloperanden in das Ziel. Das Ziel muss ein Register sein. Dieser Befehl ermöglicht, eine Adresse in einem Schritt zu errechnen, statt die Berechnung über mehrere Zwischenschritte mit den entsprechenden Operationen auszurechnen.

mov Ziel, Quelle Kopiert den Quelloperanden in den Zieloperanden. (Es ist nicht möglich gleichzeitig Ziel und Quelle als Speicherstelle anzugeben.)

rep movsb Kopiert einen Speicherbereich beginnend von der im `esi`-Register abgelegten Adresse in die im `edi`-Register abgelegten Adresse. Die zu kopierende Anzahl an Bytes wird aus dem `ecx`-Register entnommen.

sub Ziel, Quelle Subtrahiert den Quelloperanden vom Zieloperanden und speichert das Ergebnis im Zieloperanden.

and Ziel, Quelle Führt logisches UND zwischen dem Quell- und Zieloperanden und speichert das Ergebnis im Zieloperanden.

cmp Op1, Op2 Vergleich durch interne Ausführung einer Subtraktion $Op1 - Op2$. Das Ergebnis wird verworfen und Flags im `efl`-Register gesetzt, u.a. der Zero-Flag (`zf`).

jmp Label Unbedingter Sprung um eine Relativadresse.

jz Label Bedingter Sprung um eine Relativadresse, falls Zero-Flag (`zf`) gleich 1 ist.

jnz Label Bedingter Sprung um eine Relativadresse, falls Zero-Flag (`zf`) gleich 0 ist.

call Label Unterprogrammaufruf, d.h. Sprung mit Rückkehrabsicht. Vor dem Sprung wird der aktuelle Wert des Instructionpointers (`eip`-Register) auf dem Stack abgelegt und der Stackpointer (`esp`-Register) entsprechend erniedrigt.

retn Rücksprung nach dem Unterprogrammaufruf auf die Adresse, die vom Stack abgelesen wird. Danach wird der Stackpointers (`esp`-Register) entprechend erhöht.

2.4.4 ASCII-Tabelle

In Tabelle 2.5 ist ein Abschnitt der ASCII-Zeichentabelle zu finden, der sich als hilfreich für die Bearbeitung der Aufgaben erweisen könnte.

Code	Sym	Code	Sym	Code	Sym	Code	Sym	Code	Sym
20		33	3	46	*F*	59	*Y*	6C	*l*
21	!	34	4	47	*G*	5A	*Z*	6D	*m*
22	”	35	5	48	*H*	5B	[	6E	*n*
23	#	36	6	49	*I*	5C	\	6F	*o*
24	$	37	7	4A	*J*	5D	]	70	*p*
25	%	38	8	4B	*K*	5E	ˆ	71	*q*
26	&	39	9	4C	*L*	5F	_	72	*r*
27	′	3A	:	4D	*M*	60	‘	73	*s*
28	(	3B	;	4E	*N*	61	*a*	74	*t*
29	)	3C	<	4F	*O*	62	*b*	75	*u*
2A	∗	3D	=	50	*P*	63	*c*	76	*v*
2B	+	3E	>	51	*Q*	64	*d*	77	*w*
2C	,	3F	?	52	*R*	65	*e*	78	*x*
2D	−	40	@	53	*S*	66	*f*	79	*y*
2E	.	41	*A*	54	*T*	67	*g*	7A	*z*
2F	/	42	*B*	55	*U*	68	*h*	7B	–
30	0	43	*C*	56	*V*	69	*i*	7C	|
31	1	44	*D*	57	*W*	6A	*j*	7D	″
32	2	45	*E*	58	*X*	6B	*k*	7E	˜

Tabelle 2.5: Ausschnitt aus der ASCII-Zeichentabelle im Bereich 32 bis 126.

3 Musterlösungen

3.1 Lösungen zu Hauptaufgaben

Das Ziel der Hauptaufgaben war die Findung von bestimmten Eingaben, die den Fluss des untersuchten Beispielprogramms auf eine bestimmte Weise änderten.

Um alle Hauptaufgaben zu lösen, bestand die wesentliche Aufgabe in der Analyse des Stacks des laufenden Programms. Dazu sollte der Assembler-Code des Programms mithilfe von IDA durch die Anwendung der Debug-Funktion untersucht werden. Die Offenlegung des ursprünglichen C-Codes in Listing 2.1 wurde als Hilfestellung gegeben – was in der Realität normalerweise nicht der Fall ist.

In den folgenden Abschnitten wird im ersten Schritt der ursprüngliche C-Code des zu untersuchenden Beispielprogramms besprochen. Danach wird der vom C-Compiler erzeugte Assembler-Code analysiert, was die Basis für die Lösung der Aufgaben darstellt. Die Ergebnisse dieser Analyse werden dann in den anschließenden Abschnitten zur Lösungsfindung der Aufgaben herangezogen und konkrete Musterlösungen vorgestellt.

3.1.1 Ursprünglicher C-Code

Als erstes schauen wir uns den Programmablauf an. Dazu gucken wir uns erst nur den ursprünglichen C-Code in Listing 2.1 an.

Das Programm besteht nur aus der `main`-Funktion. In den Zeilen 7 bis 10 werden zunächst 4 Variablen deklariert und initialisiert. Die darauf folgenden zwei Funktionsaufrufe `printf` und `gets` fordern den Nutzer mit der Ausgabe „Enter your name (max. 9 characters):“ zur Eingabe einer Zeichenkette in der Konsole auf. Hierbei schreibt die Funktion `gets` die Eingabe des Nutzers in das C-Array `vName`. Die Variable `vName` ist ein 10-elementiges `char`-Array, also ein Array von 10 Byte-Zeichen, eine typische C-Zeichenkette. An dieser Stelle sei daran erinnert, dass das Ende einer C-Zeichenkette durch den Terminierungswert 0 gekennzeichnet wird. Folglich kann das Array `vName` höchstens eine Zeichenkette der Länge 9 speichern. In C werden Arrays auf dem Stack allokiert und die Variable `vName` ist streng genommen ein Zeiger auf den im Stack allokierten Buffer. Die Funktion `gets` bekommt also nur die Adresse des ersten Zeichens der Zeichenkette und hat keine Information über die Größe des verwendeten Buffers. Somit schreibt die Funktion `gets` die vom Nutzer in der Konsole eingegebene Zeichenkette in den Buffer auf dem Stack ohne zu prüfen, ob der Buffer für diese zu klein ist. Dies ist genau die Stelle des Beispielprogramms, an Buffer-Overflow-Angriffe stattfinden können.

Nachdem der Nutzer seine Eingabe getätigt hat, wird in Zeile 18 in einer `if`-Abfrage die Variable `var1` auf Gleichheit mit dem Hexadezimalwert 61626261 überprüft. Diese Variable wurde mit dem Dezimalwert -1 initialisiert und sonst nirgends benutzt. So sollte

diese if-Abfrage immer scheitern und bei der Ausführung der if-Rumpf nie betreten werden können.

Danach wird in Zeile 24 die Variable var2 auf Ungleichheit mit dem Hexadezimalwert 6E65706F überprüft, was immer zum Aufruf des if-Rumpfes führen sollte, da var2 mit dem Dezimalwert -256 initialisiert und sonst nirgends im Code verändert wird. Der else-Zweig sollte hier beim Ausführen also nie betreten werden. Laut Erwarten sollte also hier immer die Zeile 27 aufgerufen und durch den Aufruf der printf-Funktion der Inhalt des Arrays vName ausgegeben werden, welches die vom Nutzer eingegebene Zeichenkette beinhalten sollte.

Zum Schluss wird in Zeile 37 die Funktion getchar aufgerufen. Dieser Aufruf dient nur der Bequemlichkeit und führt dazu, dass das Konsolen-Fenster unter Windows nach Beendigung des Programms nicht sofort geschlossen wird. Für die Bearbeitung der Aufgaben hat dieser Aufruf keine Relevanz. In der darauf folgenden Zeile wird das Programm beendet und der Rückgabewert 0 zurückgegeben.

Wie wir festgestellt haben, gibt es in diesem Programm zwei Code-Abschnitte, die eigentlich nie ausgeführt werden. Folglich sollte auch die Variable pNotSoGood nie benutzt werden. Kompiliert man dieses Programm mit Optimierungen, so sollte der Compiler diesen unbenutzten Code erkennen und nicht in das Endprogramm einbinden. In unserem Fall wurde dieser C-Code ohne Optimierungen kompiliert, so dass diese Code-Abschnitte in dem Assembler-Programm enthalten sind.

3.1.2 Analyse des Assembler-Codes

Als nächstes wollen wir nun uns den generierten Assembler-Code anschauen. In Listing 3.1 ist der Assembler-Code der main-Funktion dargestellt, so wie dieser in IDA angezeigt wird. Im Folgenden gehen wir den Assembler-Code durch und stellen Verknüpfungen zum C-Code des ursprünglichen Programms in Listing 2.1 her, um nachzuvollziehen, wie die Ausführung des Programms auf Assembler-Ebene genau realisiert wird. Hierzu sollte man die Debug-Funktion von IDA einsetzen. Das Ziel bei der Betrachtung der Ausführung des Assember-Codes ist die Struktur des Stacks zu analysieren. Wir wollen vor allem herausfinden, wie und wo die einzelnen Variablen aus dem C-Code bei der Ausführung auf dem Stack abgelegt werden. Diese Information benötigen wir später, um die Werte der Variablen durch Buffer-Overflow-Attacken so verändern zu können, dass sich der Programmfluss unseren Wünschen nach ändert. Das Endergebnis der Analyse sieht man in Tabelle 3.1, die wir im Folgenden schrittweise herleiten werden. Diese Tabelle werden wir dann zur Lösung der Aufgaben heranziehen.

language

```
1 .text:004013B0 public _main
2 .text:004013B0 _main proc near     ; CODE XREF: ___mingw_CRTStartup+F8
3 .text:004013B0
4 .text:004013B0 var_3C   = dword ptr -3Ch
```

```
5 .text:004013B0 var_38   = dword ptr -38h
6 .text:004013B0 var_34   = dword ptr -34h
7 .text:004013B0 var_22   = byte ptr -22h
8 .text:004013B0 var_18   = dword ptr -18h
9 .text:004013B0 var_14   = dword ptr -14h
10 .text:004013B0 var_10   = dword ptr -10h
11 .text:004013B0
12 .text:004013B0          push  ebp
13 .text:004013B1          mov   ebp, esp
14 .text:004013B3          push  edi
15 .text:004013B4          push  esi
16 .text:004013B5          push  ebx
17 .text:004013B6          and   esp, 0FFFFFFF0h
18 .text:004013B9          sub   esp, 30h
19 .text:004013BC          call  ___main
20 .text:004013C1          mov   [esp+3Ch+var_10], offset aNotSoGood ; "not so good
    "
21 .text:004013C9          mov   [esp+3Ch+var_14], 0FFFFFF00h
22 .text:004013D1          mov   [esp+3Ch+var_18], 0FFFFFFFFh
23 .text:004013D9          lea   edx, [esp+3Ch+var_22]
24 .text:004013DD          mov   ebx, offset a123456789 ; "123456789"
25 .text:004013E2          mov   eax, 0Ah
26 .text:004013E7          mov   edi, edx
27 .text:004013E9          mov   esi, ebx
28 .text:004013EB          mov   ecx, eax
29 .text:004013ED          rep movsb
30 .text:004013EF          mov   [esp+3Ch+var_3C], offset aEnterYourNameM ; "Enter
    your name (max. 9 characters): "
31 .text:004013F6          call  _puts
32 .text:004013FB          lea   eax, [esp+3Ch+var_22]
33 .text:004013FF          mov   [esp+3Ch+var_3C], eax
34 .text:00401402          call  _gets
35 .text:00401407          cmp   [esp+3Ch+var_18], 61626261h
36 .text:0040140F          jnz   short loc_40142D
37 .text:00401411          mov   eax, [esp+3Ch+var_14]
38 .text:00401415          mov   [esp+3Ch+var_34], eax
39 .text:00401419          mov   eax, [esp+3Ch+var_18]
40 .text:0040141D          mov   [esp+3Ch+var_38], eax
41 .text:00401421          mov   [esp+3Ch+var_3C], offset aVar108xhVar208 ; "var1
    =%08xh\nvar2=%08xh\n"
42 .text:00401428          call  _printf
43 .text:0040142D
44 .text:0040142D loc_40142D:                    ; CODE XREF: _main+5F
45 .text:0040142D          cmp   [esp+3Ch+var_14], 6E65706Fh
46 .text:00401435          jz    short loc_40144D
47 .text:00401437          lea   eax, [esp+3Ch+var_22]
48 .text:0040143B          mov   [esp+3Ch+var_38], eax
49 .text:0040143F          mov   [esp+3Ch+var_3C], offset aYourNameIsS ; "Your name
     is: %s\n"
50 .text:00401446          call  _printf
```

```
51 .text:0040144B         jmp    short loc_401469
52 .text:0040144D ;
    ---------------------------------------------------------------------------
53 .text:0040144D
54 .text:0040144D loc_40144D:                   ; CODE XREF: _main+85
55 .text:0040144D         mov    eax, [esp+3Ch+var_10]
56 .text:00401451         mov    [esp+3Ch+var_34], eax
57 .text:00401455         lea    eax, [esp+3Ch+var_22]
58 .text:00401459         mov    [esp+3Ch+var_38], eax
59 .text:0040145D         mov    [esp+3Ch+var_3C], offset a_10sIsA_20sHac ; "%.10s
    is a %.20s hacker!"
60 .text:00401464         call   _printf
61 .text:00401469
62 .text:00401469 loc_401469:                   ; CODE XREF: _main+9B
63 .text:00401469         call   _getchar
64 .text:0040146E         mov    eax, 0
65 .text:00401473         lea    esp, [ebp-0Ch]
66 .text:00401476         pop    ebx
67 .text:00401477         pop    esi
68 .text:00401478         pop    edi
69 .text:00401479         pop    ebp
70 .text:0040147A         retn
71 .text:0040147A _main endp
```

Listing 3.1: Assembler-Code der main-Funktion in IDA.

Auf der linken Seite des Assembler-Codes wird das Segment angegeben. Bei `.text` handelt es sich um das Code-Segment, d.h. in diesem Speicherbereich befindet sich der ausführbare Code des Programms. Nach dem Doppelpunkt folgt die Speicheradresse.

In den Zeilen 4 bis 10 sind die Assembler-Variablen notiert. Es sind Offsets, die für die Adressierung des Stacks benötigt werden.

Ab Zeile 12 beginnt die Ausführung des eigentlichen Programm-Codes der Methode `main`. Zur Analyse in IDA können wir an diese Zeile einen Breakpoint setzen und das Programm im Debug-Modus starten. Angehalten am Breakpoint, interessiert uns als erstes der Stackpointer, also der Inhalt des `esp`-Registers. Dieser zeigt auf das zuletzt eingefügte Element im Stack. Da unmittelbar vor dem Breakpoint die Funktion `main` aufgerufen wurde, sollte auf dem Stack die Rücksprungadresse liegen. In `esp` steht die Adresse 0022FF5C, dies ist die Spitze des aktuellen Stacks, in der die Adresse 004010FD gespeichert ist. Ein Blick auf diese Adresse in IDA zeigt uns, dass es sich tatsächlich um die Rücksprungadresse handelt. Denn auf der Adresse 004010F8 befindet sich der Befehl `call _main`. Dieser ruft die Methode `main` auf und legt die Adresse des nächsten Befehls, der nach der Ausführung der `main`-Funktion ausgeführt werden soll, auf dem Stack ab, also die Adresse 004010FD. Diese Adresse ist für die spätere Bearbeitung der Aufgaben wichtig und somit tragen wir sie in die Tabelle 3.1 ein (die unterste Zeile).

Stack				Adresse	Kommentar
				10	Argumente von Funktionsaufrufen aus `main`.
				14	Argumente von Funktionsaufrufen aus `main`.
				18	Argumente von Funktionsaufrufen aus `main`.
—				$1C$	Nicht benutzt.
—				20	Nicht benutzt.
—				24	Nicht benutzt.
'2'	'1'	—		28	Zeichenketten-Buffer `vName`.
'6'	'5'	'4'	'3'	$2C$	Zeichenketten-Buffer `vName`.
00	'9'	'8'	'7'	30	Zeichenketten-Buffer `vName`.
FF	FF	FF	FF	34	Integer-Variable `var1`.
FF	FF	FF	00	38	Integer-Variable `var2`.
00	40	30	64	$3C$	Zeichenketten-Zeiger `pNotSoGood`.
—				40	Nicht benutzt.
—				44	Nicht benutzt.
—				48	Nicht benutzt.
ebx				$4C$	`ebx`-Registerwert vor dem Aufruf von `main`.
esi				50	`esi`-Registerwert vor dem Aufruf von `main`.
edi				54	`edi`-Registerwert vor dem Aufruf von `main`.
ebp				58	Basepointer vor dem Aufruf von `main`.
00	40	10	FD	$5C$	Vom Befehl `call _main` abgelegte Rücksprungadresse.

Tabelle 3.1: Analyse des Stacks. In den Speicherstellen der Variablen befinden sich die jeweiligen Initialisierungswerte.

Bei den nächsten Befehlen in den Zeilen 12 bis 19 handelt es sich um Vorbereitungen des Stacks, bevor der eigentliche Rumpf-Code der Funktion `main` beginnt. Hier wird der neue Stackrahmen erzeugt und einige nicht-flüchtige Register auf den Stack gelegt, die in der `main`-Funktion benutzt werden. Nach der Ausführung dieser Befehle, liegen auf dem Stack die Inhalte der Register `ebp`, `edi`, `esi` und `ebx` und der Stackpointer zeigt auf die Spitze des neuen Stackrahmens. Die Inhalte der gespeicherten Register haben uns im Folgenden nicht zu interessieren, so vermerken wir in Tabelle 3.1 nur ihre Positionen im Stack. Der Stackrahmen wird auf 48 Byte gesetzt. Dies sehen wir an der neu gesetzten Adresse des Stackpointers, welche nun 0022FF10 lautet. Diese Adresse legt auch die Größe unserer Tabelle 3.1 fest – alles, was außerhalb dieses Stackrahmens liegt, ist für uns nicht relevant.

Als nächstes werden in den Zeilen 20 bis 29 die Variablen aus dem C-Code auf dem Stack initialisiert. Dies ist für uns der interessanteste Abschnitt im Assembler-Code, da uns dieser Einsicht gibt, wo genau auf dem Stack die einzelnen Variablen abgelegt werden. Initialisiert werden diese in der gleichen Reihenfolge wie im ursprünglichen C-Code angegeben (Zeilen 7 bis 10 in Listing 2.1). Zuerst wird der Zeiger `pNotSoGood` initialisiert. Dazu wird mit dem `mov`-Befehl die Adresse der Zeichenkette `"not so good"` auf den Stack geschrieben, und zwar auf die Adresse 0022FF3C. An dieser Stelle sei betont, dass es sich bei `pNotSoGood` lediglich um einen Zeiger auf eine Zeichenkette handelt im

Gegensatz zu `vName`, was ein Array darstellt. Die Zeichenkette `"not so good"` liegt in dem Read-Only-Segment des Speichers auf der Adresse 00403064. Diese notieren wir entsprechend in Tabelle 3.1. Desweiteren wird auf die Adresse 0022FF38 der Wert FFFFFF00 und auf die Adresse 0022FF34 der Wert FFFFFFFF geschrieben. Ersteres entspricht dem dezimalen Wert -256 und, wie dem C-Code entnommen werden kann, also der Variable `var2`, letzteres entspricht dem dezimalen Wert -1, also der Variable `var1`. Alle drei Variablen werden direkt nebeneinander auf dem Stack platziert, siehe Tabelle 3.1. Als letztes wird der Buffer, also das C-Array `vName` initialisiert. Hierzu wird die Zeichenkette `"123456789"` mit dem Befehl `rep movsb` in Zeile 29 auf den Stack geschrieben. Dieser Befehl wird parametrisiert durch Register. In `edi` wird 0022FF2A als Zieladresse geschrieben. In `esi` wird 00403038 als Quelladresse geschrieben. Wie man in IDA nachgucken kann, handelt es sich bei der letzteren Adresse um die Speicherstelle der Zeichenkette `"123456789"`. In `ecx` wird der Wert 10 geschrieben, dies ist die Länge der zu kopierenden Zeichenkette. Nach Ausführung des Befehls `rep movsb` erscheint in IDA die Zeichenkette auf dem Stack, so wie in Tabelle 3.1 notiert. An letzter Stelle des Buffers (Adresse 0022FF33) wird eine terminierende Null geschrieben.

Die nächsten Assembler-Code-Zeilen 30 bis 34 entsprechen den C-Code-Zeilen 12 bis 16. Diesem Assembler-Code können wir entnehmen, dass die Speicherstellen auf der Spitze des Stacks zur Ablegung der Argumente für Funktionsaufrufe `printf` und `gets` benutzt werden (siehe obere Zeilen in Tabelle 3.1). Das Ausschlaggebende für uns geschieht in Zeile 34 beim Aufruf der Funktion `gets`. Hier wird der Benutzer zur Eingabe aufgefordert und diese wird dann in den Buffer geschrieben. An dieser Stelle können wir experimentieren, was für Auswirkungen die Nutzereingabe auf den Stack hat. Wenn der Nutzer in der Konsole wie folgt *Denis* eingibt, so wird der Stack wie in Tabelle 3.2 bzw. 3.3 gefüllt.

```
Enter your name (max. 9 characters):
Denis
Your name is: Denis
```

Werden mehr als 9 Zeichen eingegeben, so kommt es zu einem Buffer-Overflow und die überlaufenden Zeichen überschreiben die Werte der darauf folgenden Speicherstellen im Stack, die u.a. für die Variablen `var2`, `var1` und `pNotSoGood` reserviert wurden, so wie wir es in Tabelle 3.1 notiert haben. Dieses Verhalten werden wir später gezielt ausnutzen, um Buffer-Overflow-Attacken durchzuführen.

Die Zeilen 35 bis 42 entsprechen dem ersten `if`-Konstrukt im C-Code. Wie wir in IDA verfolgen können, wird hier mit dem Befehl `cmp` der Wert vom Stack auf der Speicherstelle 0022FF34 mit dem festen Wert 61626261 verglichen. Unserer Analyse (Tabelle 3.1) und dem ursprünglichen C-Code können wir entnehmen, dass die Speicherstelle 0022FF34 der Variable `var1` entspricht. Durch den darauf folgenden Befehl `jnz` wird das Ergebnis des `cmp`-Vergleichs aufgegriffen und der Programmfluss entsprechend bestimmt.

Stack				Adresse	Kommentar
'e'	'D'	—		28	Zeichenketten-Buffer `vName`.
00	's'	'i'	'n'	2C	Zeichenketten-Buffer `vName`.
00	'9'	'8'	'7'	30	Zeichenketten-Buffer `vName`.
FF	FF	FF	FF	34	Integer-Variable `var1`.
FF	FF	FF	00	38	Integer-Variable `var2`.
00	40	30	64	3C	Zeichenketten-Zeiger `pNotSoGood`.

Tabelle 3.2: Stack nach Eingabe der Zeichenkette *Denis* dargestellt als Zeichensymbole.

Stack				Adresse	Kommentar
65	44	—		28	Zeichenketten-Buffer `vName`.
00	73	69	6E	2C	Zeichenketten-Buffer `vName`.
00	39	38	37	30	Zeichenketten-Buffer `vName`.
FF	FF	FF	FF	34	Integer-Variable `var1`.
FF	FF	FF	00	38	Integer-Variable `var2`.
00	40	30	64	3C	Zeichenketten-Zeiger `pNotSoGood`.

Tabelle 3.3: Stack nach Eingabe der Zeichenkette *Denis* dargestellt mit Hexadezimalwerten.

Das `if-else`-Konstrukt aus dem C-Code entspricht den Assembler-Code-Zeilen 45 bis 60. Hier wird mit dem `cmp`-Befehl der Wert vom Stack auf der Speicherstelle 0022FF38, also der Inhalt der Variable `var2`, mit dem festen Wert 6E65706F verglichen.

Zum Schluss wird in den Zeilen 64 bis 70 der Zustand der Register wiederhergestellt und durch den Aufruf des Befehls `retn` die `main`-Funktion verlassen durch das Springen auf die Rücksprungadresse, welche im Stack auf der Speicherstelle 0022FF5C abgelegt wurde.

3.1.3 Lösungen zu Aufgabe 1

Finde eine Eingabe, so dass der Inhalt der C-Variablen `var1` *und* `var2` *ausgegeben wird.*

In dieser Aufgabe war das Ziel, den Programmfluss durch eine Eingabe so zu ändern, dass der `if`-Rumpf in Zeile 22 des C-Codes in Listing 2.1 ausgeführt wird. Dies entspricht den Assembler-Code-Zeilen 37 bis 42 in Listing 3.1. Dazu muss der Inhalt der C-Variable `var1` auf den Hexadezimalwert 61626261 gesetzt werden. Nach der Analyse des Assembler-Codes können wir der Tabelle 3.1 entnehmen, dass die Variable `var1` sich direkt nach dem Array `vName` im Stack befindet. Das führt dazu, dass nach dem 10. Zeichen der Nutzer-Eingabe, die Zeichen 11 bis einschließlich 14 direkt den Wert der Variable `var1` ändern. Wie man der Tabelle 2.5 entnehmen kann, entspricht der Hexadezimalwert 61626261 genau der Zeichenkette *abba*.

Als eine mögliche Lösung folgt:

```
Enter your name (max. 9 characters):
```

```
Denis Grafabba
var1=61626261h
var2=ffffff00h
Your name is: Denis Grafabba
```

Der Zustand des Stacks unmittelbar nach dieser Eingabe kann den Tabellen 3.4 und 3.5 entnommen werden. Hierbei sei zu beachten, dass die terminierende Null auf die Speicherstelle 0022FF38 geschrieben wird, welche der Variable `var2` zugeordnet ist. Dies kann leicht übersehen werden, da das Beispiel durch die Initialisierung der Variable `var2` mit dem Wert -256 extra so konstruiert wurde, dass an diese Stelle von Anfang an der Wert 0 geschrieben wird.

Stack				Adresse	Kommentar
'e'	'D'	—		28	Zeichenketten-Buffer `vName`.
' '	's'	'i'	'n'	2C	Zeichenketten-Buffer `vName`.
'f'	'a'	'r'	'G'	30	Zeichenketten-Buffer `vName`.
'a'	'b'	'b'	'a'	34	Integer-Variable `var1`.
FF	FF	FF	00	38	Integer-Variable `var2`.
00	40	30	64	3C	Zeichenketten-Zeiger `pNotSoGood`.

Tabelle 3.4: Stack nach Eingabe der Zeichenkette *Denis Grafabba* dargestellt als Zeichensymbole.

Stack				Adresse	Kommentar
65	44	—		28	Zeichenketten-Buffer `vName`.
20	73	69	6E	2C	Zeichenketten-Buffer `vName`.
66	61	72	47	30	Zeichenketten-Buffer `vName`.
61	62	62	61	34	Integer-Variable `var1`.
FF	FF	FF	00	38	Integer-Variable `var2`.
00	40	30	64	3C	Zeichenketten-Zeiger `pNotSoGood`.

Tabelle 3.5: Stack nach Eingabe der Zeichenkette *Denis Grafabba* dargestellt mit Hexadezimalwerten.

3.1.4 Lösungen zu Aufgabe 2

Finde eine Eingabe, so dass „[Your name] is a not so good hacker!“ ausgegeben wird.

In dieser Aufgabe sollte zunächst nach der Analogie zu Aufgabe 1 verfahren werden. Beim Anschauen des ursprünglichen C-Codes in Listing 2.1, sollte festgestellt werden, dass zur Lösung der Aufgabe der `else`-Rumpf in Zeile 32 ausgeführt werden sollte. Dies entpricht den Assembler-Code-Zeilen 55 bis 60 in Listing 3.1. Dazu muss der Inhalt der C-Variable `var2` auf den Hexadezimalwert 6E65706F gesetzt werden. Nach der Analyse des Assembler-Codes können wir der Tabelle 3.1 entnehmen, dass die Variable `var2` sich

direkt nach der Variable `var1` auf dem Stack befindet. Das führt dazu, dass nach dem 14. Zeichen der Nutzer-Eingabe, die Zeichen 15 bis einschließlich 19 direkt den Wert der Variable `var2` ändern. Wie man der Tabelle 2.5 entnehmen kann, entspricht der Hexadezimalwert 6E65706F genau der Zeichenkette *nepo*. Wie wir im folgenden sehen, führt die Eingabe der so konstruierten Zeichenkette nicht zu dem gewünschten Ergebnis:

```
Enter your name (max. 9 characters):
Denis Grafabbanepo
var1=61626261h
var2=6f70656eh
Your name is: Denis Grafabbanepo
```

Der Konsolen-Ausgabe des Inhaltes der Variable `var2` kann entnommen werden, dass der Fehler in der falschen Reihenfolge der Zeichen liegt. Die Zeichenkette, die wir eigentlich suchen, lautet also *open*. Doch auch diese Eingabe sollte zu einem verwirrenden Ergebnis führen:

```
Enter your name (max. 9 characters):
Denis Grafabbaopen
var1=61626261h
var2=6e65706fh
Denis Graf is a libgcc_s_dw2-1.dll hacker!
```

Der `else`-Rumpf wurde wie gewünscht ausgeführt, doch anstelle der Zeichenkette `"not so good"` wurde `"libgcc_s_dw2-1.dll"` ausgegeben. Der Blick in IDA auf den Stack unmittelbar nach der Eingabe bringt Licht in das Rätsel. Der Inhalt des Stacks ist in den Tabellen 3.6 und 3.7 dargestellt. Das Problem besteht darin, dass der Inhalt des C-Zeigers `pNotSoGood` verändert wurde, da dieser direkt nach der C-Variable `var2` auf dem Stack abgelegt ist. Die terminierende Null ändert den Zeiger-Wert von 00403064 auf 00403000. Der Blick beim Debuggen in IDA auf die Adresse 00403000 zeigt, dass an dieser Speicherstelle „zufälligerweise" die Zeichenkette `"libgcc_s_dw2-1.dll"` gespeichert ist. Dies führt zu dieser mysteriösen Ausgabe in der Konsole.

Stack				Adresse	Kommentar
'e'	'D'	—		28	Zeichenketten-Buffer `vName`.
' '	's'	'i'	'n'	2C	Zeichenketten-Buffer `vName`.
'f'	'a'	'r'	'G'	30	Zeichenketten-Buffer `vName`.
'a'	'b'	'b'	'a'	34	Integer-Variable `var1`.
'n'	'e'	'p'	'o'	38	Integer-Variable `var2`.
00	40	30	00	3C	Zeichenketten-Zeiger `pNotSoGood`.

Tabelle 3.6: Stack nach Eingabe der Zeichenkette *Denis Grafabbaopen* dargestellt als Zeichensymbole.

Stack				Adresse	Kommentar
65	44	—		28	Zeichenketten-Buffer `vName`.
20	73	69	6E	2*C*	Zeichenketten-Buffer `vName`.
66	61	72	47	30	Zeichenketten-Buffer `vName`.
61	62	62	61	34	Integer-Variable `var1`.
6E	65	70	6F	38	Integer-Variable `var2`.
00	40	30	00	3*C*	Zeichenketten-Zeiger `pNotSoGood`.

Tabelle 3.7: Stack nach Eingabe der Zeichenkette *Denis Grafabbaopen* dargestellt mit Hexadezimalwerten.

Da wir es nicht umgehen können, dass die terminierende Null den Zeiger-Wert von `pNotSoGood` ändert, besteht die Lösung darin, diesen Wert wiederherzustellen. Dazu erweitern wir die Eingabe, so dass 00403064 ins `pNotSoGood` geschrieben wird. Dies entspricht der Zeichenkette *opend0@*. Hierbei passt es „zufällig“, dass die terminierende Null als letztes Zeichen auf die Speicheradresse 0022FF3F geschrieben wird.

Als eine mögliche Lösung folgt:

```
Enter your name (max. 9 characters):
Denis Grafabbaopend0@
var1=61626261h
var2=6e65706fh
Denis Graf is a not so good hacker!
```

Der Zustand des Stacks unmittelbar nach dieser Eingabe kann den Tabellen 3.8 und 3.9 entnommen werden.

Stack				Adresse	Kommentar
’e’	’D’	—		28	Zeichenketten-Buffer `vName`.
’ ’	’s’	’i’	’n’	2*C*	Zeichenketten-Buffer `vName`.
’f’	’a’	’r’	’G’	30	Zeichenketten-Buffer `vName`.
’a’	’b’	’b’	’a’	34	Integer-Variable `var1`.
’n’	’e’	’p’	’o’	38	Integer-Variable `var2`.
00	’0’	’@’	’d’	3*C*	Zeichenketten-Zeiger `pNotSoGood`.

Tabelle 3.8: Stack nach Eingabe der Zeichenkette *Denis Grafabbaopend0@* dargestellt als Zeichensymbole.

3.1.5 Lösungen zu Aufgabe 3

Finde eine Eingabe, so dass „[Your name] is a good hacker!“ ausgegeben wird.

Die Lösung dieser Aufgabe baut auf Aufgabe 2 auf. Hierzu sollten also die beschriebenen Fehler bei der Lösung von Aufgabe 2 begangen und nachvollzogen worden sein. Da bei der Ausgabe auf den Zeiger `pNotSoGood` zugegriffen wird, besteht die Lösung darin,

Stack				Adresse	Kommentar
65	44	—		28	Zeichenketten-Buffer `vName`.
20	73	69	6E	2*C*	Zeichenketten-Buffer `vName`.
66	61	72	47	30	Zeichenketten-Buffer `vName`.
61	62	62	61	34	Integer-Variable `var1`.
6E	65	70	6F	38	Integer-Variable `var2`.
00	40	30	64	3*C*	Zeichenketten-Zeiger `pNotSoGood`.

Tabelle 3.9: Stack nach Eingabe der Zeichenkette *Denis Grafabbaopend0@* dargestellt mit Hexadezimalwerten.

den Wert von diesem so zu manipulieren, dass er nicht auf den Anfang der Zeichenkette `"not so good"` zeigt, sondern auf die Teilzeichenkette `"good"`. Hierzu muss die ursprüngliche Adresse 00403064 um 7 Stellen verschoben werden, also auf den Wert 0040306B. Somit benutzen wir anstelle des Zeichens *d* das Zeichen *k*. So müssen die Zeichen 15 bis 21 der Zeichenkette *openk0@* entsprechen.

Als eine mögliche Lösung folgt:

```
Enter your name (max. 9 characters):
Denis Graf    openk0@
Denis Graf is a good hacker!
```

Der Zustand des Stacks unmittelbar nach dieser Eingabe kann den Tabellen 3.10 und 3.11 entnommen werden.

Stack				Adresse	Kommentar
'e'	'D'	—		28	Zeichenketten-Buffer `vName`.
' '	's'	'i'	'n'	2*C*	Zeichenketten-Buffer `vName`.
'f'	'a'	'r'	'G'	30	Zeichenketten-Buffer `vName`.
' '	' '	' '	' '	34	Integer-Variable `var1`.
'n'	'e'	'p'	'o'	38	Integer-Variable `var2`.
00	'0'	'@'	'k'	3*C*	Zeichenketten-Zeiger `pNotSoGood`.

Tabelle 3.10: Stack nach Eingabe der Zeichenkette *Denis Graf   openk@0* dargestellt als Zeichensymbole.

Stack				Adresse	Kommentar
65	44	—		28	Zeichenketten-Buffer `vName`.
20	73	69	6E	2*C*	Zeichenketten-Buffer `vName`.
66	61	72	47	30	Zeichenketten-Buffer `vName`.
20	20	20	20	34	Integer-Variable `var1`.
6E	65	70	6F	38	Integer-Variable `var2`.
00	40	30	6B	3*C*	Zeichenketten-Zeiger `pNotSoGood`.

Tabelle 3.11: Stack nach Eingabe der Zeichenkette *Denis Graf   openk@0* dargestellt mit Hexadezimalwerten.

3.1.6 Lösungen zu Aufgabe 4

Finde eine Eingabe, so dass der Aufruf der `main`-Methode neugestartet wird und der Benutzer erneut zur Eingabe seines Namens aufgefordert wird.

In dieser Aufgabe besteht die Lösung darin, die Rücksprungadresse zu manipulieren. Nach der Analyse des Assembler-Codes wissen wir, dass die Rücksprungadresse auf der Speicherstelle 0022FF5C auf dem Stack gespeichert wird. Am Ende der `main`-Funktion wird diese vom Befehl `retn` ausgelesen und zum Rücksprung benutzt. Der Tabelle 3.1 können wir die ursprüngliche Adresse entnehmen. Diese lautet 004010FD. Die Funktion `main` wird von der Adresse 004010F8 mit dem `call`-Befehl aufgerufen. Als Lösung liegt es folglich nahe, den Wert auf dem Stack auf diese Adresse zu überschreiben. Doch die Hexadezimalwerte 10 und F8 liegen außerhalb des ASCII-Zeichen-Bereichs und können nicht direkt in der Konsole eingegeben werden. Dieses Problem sollte umgangen werden, indem die Rücksprungadresse nicht exakt auf die Adresse des `call`-Befehs gesetzt wird, sondern auf eine Adresse, welche vor diesem liegt. So kann die Aufgabe gelöst werden, indem durch die terminierende Null der Konsolen-Eingabe nur das niedere Byte der Rücksprungadresse geändert wird, also auf den Wert 00401000. Der Tabelle 3.1 können wir entnehmen, dass dazu eine beliebige Zeichenkette aus genau 50 Zeichen eingegeben werden muss.

3.2 Lösungen zu Zusatzaufgaben

3.2.1 Lösungen zu Aufgabe 1

Funktionieren noch die Buffer-Overflow-Angriffe aus den Hauptaufgaben?

Keine der Angriffseingaben aus den Hauptaufgaben funktionieren nun.

3.2.2 Lösungen zu Aufgabe 2

Was wurde im Assembler-Code verändert?

Die wesentlichen Unterschiede:

- Die Reihenfolge der Variablen wurde vertauscht.
- Das Array `vName` befindet sich nun nicht vor sondern nach den anderen C-Variablen auf Stack.
- Am Anfang der `main`-Funktion wurde Code zum Setzen des Cannarys und am Ende der `main`-Funktion Code zum Prüfen des Cannary-Wertes hinzugefügt.

3.2.3 Lösungen zu Aufgabe 3

Auf welche Speicherstelle wurde das Cannary gesetzt und mit welchem Wert initialisiert?

Das Cannary wurde auf den Hexadezimalwert FF0A0000 gesetzt. Der Tabelle 3.12 kann die Platzierung der Variablen und des Cannarys entnommen werden.

Stack				Adresse	Kommentar
00	40	30	64	24	Zeichenketten-Zeiger `pNotSoGood`.
FF	FF	FF	00	28	Integer-Variable `var2`.
FF	FF	FF	FF	2C	Integer-Variable `var1`.
′2′	′1′	—		30	Zeichenketten-Buffer `vName`.
′6′	′5′	′4′	′3′	34	Zeichenketten-Buffer `vName`.
00	′9′	′8′	′7′	38	Zeichenketten-Buffer `vName`.
FF	0A	00	00	3C	Canary.

Tabelle 3.12: Stack nach Kompilieren mit dem Flag *-fstack-protector-all*. In den Speicherstellen der Variablen befinden sich die jeweiligen Initialisierungswerte.

3.2.4 Lösungen zu Aufgabe 4

Ist es dennoch möglich diesen Schutz irgendwie zu umgehen, um einige der in den Hauptaufgaben gesetzten Angriffe zu erzielen?

Da das Array `vName` nun nach den anderen Variablen im Stack platziert wird, kann ein Buffer-Overflow diese nie überschreiben. Stattdessen führt jedes Buffer-Overflow zum Überschreiben des Cannary-Wertes, da das Cannary direkt nach dem Array `vName` platziert wird. Das Cannary wird am Ende der `main`-Funktion auf seinen ursprünglichen Wert überprüft und das Programm abgebrochen, falls dieser sich verändert hat. Der einzige Angriffsansatz bestünde darin, mit einem Buffer-Overflow den Cannary-Wert wiederherzustellen und die Rücksprungadresse zu überschreiben. Dies ist aber nicht durch die Konsolen-Eingabe möglich, da der für das Cannary eingesetzte Wert nicht eingebbar ist.

Literaturverzeichnis

[1] J. Deckard. *Buffer Overflow Attacks: Detect, Exploit, Prevent*. Elsevier Science, 2005.

[2] Robert Fithen, William L.; Seacord. Vt-mb. violation of memory bounds. 2007.

[3] H. Herold. *Linux/Unix-Systemprogrammierung*. Open source library. Addison-Wesley, 2004.

[4] Tobias Klein. *Buffer Overflows und Format-String-Schwachstellen*. Dpunkt.Verlag GmbH, 2003.

[5] Stephen Checkoway; Damon McCoy; Brian Kantor; Danny Anderson; Hovav Shacham; Stefan Savage; Karl Koscher; Alexei Czeskis; Franziska Roesner and Tadayoshi Kohno. *Comprehensive Experimental Analyses of Automotive Attack Surfaces*. 2011.

[6] Ryan Russell, Rain Forest Puppy, and Mudge. *Hack Proofing Your Network*. San Val, 2001.

[7] Theo de Raadt Todd C. Miller. strlcpy and strlcat - consistent, safe, string copy and concatenation. 1999.