

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTEL尼亚

FRAGMENTY KSIĄŻEK ONLINE

Strażnik bezpieczeństwa danych

Autorzy: Cyrus Peikari, Anton Chuvakin

Tłumaczenie: Witold Ziolo (przedmowa, rozdz. 1 – 10),
Marek Pętlicki (rozdz. 11 – 17), Sławomir Dzieniszewski
(rozdz. 18 – 22, dod. A)

ISBN: 83-7361-513-X

Tytuł oryginału: [Security Warrior](#)

Format: B5, stron: 606



Poznaj swojego napastnika

Nietrudno znaleźć książkę o bezpieczeństwie komputerów – wielu użytkowników i administratorów czuje się przytłoczonych liczbą tego rodzaju publikacji. Jednak z każdym nowym programem wykorzystującym słabości systemów komputerowych wzrasta stopień wyrafinowania ataków na nie.

Każdy administrator musi być świadomy różnego rodzaju zagrożeń związanych z jego komputerem, jak np. istnienie trojanów – niewinnie wyglądających programów szpiegowskich wysyłających informacje o poczynaniach użytkownika, wykorzystanie reinżynierii oprogramowania czy ataków z użyciem SQL.

„Strażnik bezpieczeństwa danych” mówi o tym, co najgorszego może spotkać użytkownika komputera. Jej autorzy wyznają zasadę, że aby skutecznie się bronić, należy jak najlepiej zrozumieć napastnika. Obejmuje ona szeroki zakres tematów, począwszy od reinżynierii oprogramowania, poprzez ataki na systemy operacyjne, a skończywszy na socjotechnice – wszystko po to, aby poznać wroga i przygotować się do walki.

„Strażnik bezpieczeństwa danych” jest najbardziej wyczerpującą i najbardziej aktualną książką opisującą sztukę wirtualnych wojen – ataków wymierzonych w systemy komputerowe oraz obrony przed nimi.

O autorach:

Doktor Cyrus Peikari jest założycielem firmy Airscanner Corporation, producenta narzędzi do zabezpieczania sieci bezprzewodowych. Jest autorem kilku książek poświęconych bezpieczeństwu systemów komputerowych i często bierze czynny udział w konferencjach dotyczących tego zagadnienia. Zajmuje się również zabezpieczeniami w InformIT.com.

Doktor Anton Chuvakin jest starszym analitykiem do spraw zabezpieczeń, pracuje w firmie netForensics. Specjalizuje się w wykrywaniu włamań, zabezpieczaniu sieci i systemów oraz wykrywaniu ich słabych punktów. Jest autorem wielu artykułów dotyczących zabezpieczania sieci i komputerów.

Jeżeli znajdujesz się na linii frontu, broniąc swoich sieci przed atakami, na pewno będziesz potrzebował tej książki.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl



Spis treści

<i>Przedmowa</i>	<i>11</i>
<i>Część I Łamanie programów</i>	<i>17</i>
<i>Rozdział 1. Asembler.....</i>	<i>19</i>
Rejestry	20
Rozkazy asemblera	23
Źródła dodatkowych informacji.....	24
<i>Rozdział 2. Reinżynieria oprogramowania systemu Windows</i>	<i>25</i>
Historia reinżynierii oprogramowania	27
Narzędzia reinżynierii oprogramowania	27
Przykłady reinżynierii oprogramowania.....	39
Źródła dodatkowych informacji.....	47
<i>Rozdział 3. Reinżynieria oprogramowania systemu Linux.....</i>	<i>49</i>
Podstawowe narzędzia i techniki	50
Dobra deasemblacja.....	71
Trudności	84
Pisanie nowych narzędzi	89
Źródła dodatkowych informacji.....	129
<i>Rozdział 4. Reinżynieria oprogramowania systemu Windows CE</i>	<i>131</i>
Architektura systemu Windows CE	132
Podstawy reinżynierii oprogramowania systemu Windows CE.....	137
Reinżynieria oprogramowania Windows CE w praktyce	145
Reinżynieria programu serial.exe	161
Źródła dodatkowych informacji.....	174

Rozdział 5. Ataki przepełnienia bufora	175
Przepełnienie bufora	175
Wyjaśnienie pojęcia bufora	177
Uderzenie w stos.....	179
Przepełnienie sterty	180
Zapobieganie przepełnieniom bufora	180
Przepełnienie bufora w praktyce	182
Źródła dodatkowych informacji.....	189
 Część II Polowanie w sieci	191
 Rozdział 6. Protokoły TCP/IP	193
Krótka historia TCP/IP.....	193
Enkapsulacja	194
Protokół TCP	195
Protokół IP	197
Protokół UDP	199
Protokół ICMP.....	199
Protokół ARP	200
Protokół RARP	200
Protokół BOOTP	201
Protokół DHCP	201
Uzgadnianie TCP/IP.....	201
Ukryte kanały	203
Protokół IPv6	203
Ethereal	205
Analiza pakietów	206
Fragmentacja.....	208
Źródła dodatkowych informacji.....	214
 Rozdział 7. Socjotechnika	215
Podstawy	216
Prowadzenie ataków	218
Zaawansowana socjotechnika.....	226
Źródła dodatkowych informacji.....	228
 Rozdział 8. Rekonesans	229
Rekonesans online	229
Wnioski.....	242
Źródła dodatkowych informacji.....	242

Rozdział 9. Rozpoznawanie systemów operacyjnych	243
Ustanowienie sesji telnet	243
Rozpoznawanie stosu TCP	244
Narzędzia specjalizowane	247
Pasywne rozpoznawanie systemów operacyjnych	247
Nieostre rozpoznawanie systemów operacyjnych	251
Rozpoznawanie systemu operacyjnego na podstawie wartości czasu oczekiwania TCP/IP	253
Źródła dodatkowych informacji	254
Rozdział 10. Zacieranie śladów	255
Przed kim należy się ukrywać?	255
Zacieranie śladów po ataku	256
Ukrywanie się przed środkami dochodzeniowymi	262
Utrzymanie dostępu	268
Źródła dodatkowych informacji	275
Część III Ataki na systemy	277
Rozdział 11. Obrona systemu Unix	279
Hasła	280
Dostęp do plików	284
Dzienniki systemowe	287
Sieciowy dostęp do systemów Unix	291
Wzmacnianie Uniksa	296
Obrona sieci uniksowej	313
Serwer Apache	319
Źródła dodatkowych informacji	328
Rozdział 12. Ataki na system Unix	331
Ataki lokalne	331
Ataki zdalne	340
Ataki blokady usług	357
Źródła dodatkowych informacji	365
Rozdział 13. Ataki na systemy klienckie Windows	367
Ataki blokady usług	367
Ataki zdalne	377
Zdalny pulpit i zdalna pomoc	380
Źródła dodatkowych informacji	385

Rozdział 14. Ataki na serwery Windows.....	387
Historia wersji	387
Ataki na mechanizm uwierzytelniania Kerberos	388
Omijanie ochrony przed przepełnieniem bufora	393
Słabości mechanizmu Active Directory	394
Łamanie zabezpieczeń PKI	396
Łamanie zabezpieczeń kart elektronicznych.....	397
Szyfrowany system plików	400
Zewnętrzne narzędzia szyfrujące.....	402
Źródła dodatkowych informacji.....	405
 Rozdział 15. Bezpieczeństwo usług WWW opartych na protokole SOAP.....	407
Szyfrowanie XML	408
Podpisy cyfrowe XML	410
Źródła dodatkowych informacji.....	411
 Rozdział 16. Ataki za pomocą wymuszeń kodu SQL.....	413
Wprowadzenie do języka SQL	413
Ataki wymuszeń kodu SQL	417
Obrona przed wymuszeniami kodu SQL	424
Przykłady PHP-Nuke.....	428
Źródła dodatkowych informacji.....	431
 Rozdział 17. Bezpieczeństwo sieci bezprzewodowych	433
Redukcja rozproszenia sygnału	433
Problemy z WEP	435
Łamanie zabezpieczeń WEP	435
Łamanie zabezpieczeń WEP w praktyce	441
Sieci VPN.....	442
TKIP	443
SSL.....	444
Wirusy bezprzewodowe.....	444
Źródła dodatkowych informacji.....	449
 Część IV Zaawansowane techniki obrony	451
 Rozdział 18. Analiza śladów w systemie audytowania	453
Podstawy analizy dzienników.....	454
Przykłady dzienników	454
Stany rejestrowane w dziennikach	464

Kiedy zagłądać do dzienników?	465
Przepełnienie dzienników i ich agregacja.....	467
Wyzwania związane z analizą dzienników	468
Narzędzia do zarządzania informacjami bezpieczeństwa	469
Globalna agregacja dzienników	469
Źródła dodatkowych informacji.....	470
Rozdział 19. Systemy detekcji włamań	471
Przykłady systemów IDS	472
Zastosowanie analizy Bayesowskiej	478
Sposoby oszukiwania systemów IDS	484
Przyszłość systemów IDS	486
Przypadek systemu IDS Snort.....	489
Błędy związane z rozmieszczaniem systemów IDS.....	494
Źródła dodatkowych informacji.....	496
Rozdział 20. Słoiki miodu	497
Motywy	499
Budowanie infrastruktury	499
Przechwytywanie ataków	511
Źródła dodatkowych informacji.....	512
Rozdział 21. Reagowanie na incydenty.....	513
Przykład z życia: chaos z powodu robaka	513
Definicje.....	514
Podstawowy schemat reagowania na incydenty	517
Małe sieci.....	521
Sieci średnich rozmiarów	528
Duże sieci	530
Źródła dodatkowych informacji.....	535
Rozdział 22. Zbieranie i zacieranie śladów.....	537
Podstawowe informacje o komputerach.....	538
Odpadki informacji	540
Narzędzia do zbierania śladów	541
Narzędzia śledcze na rozruchowych dyskach CD-ROM.....	547
Evidence Eliminator	551
Przykład procedury zbierania dowodów: atak na serwer FTP.....	559
Źródła dodatkowych informacji.....	569

<i>Dodatki</i>	571
<i>Dodatek A Przydatne polecenia SoftICE i lista punktów wstrzymania</i>	573
Polecenia SoftICE	573
Kody wstrzymania	577
<i>Skorowidz</i>	581

5

Ataki przepełnienia bufora

Atakowanie aplikacji jest podstawową techniką, którą stosują inżynierowie badający słabe punkty programów. W ten sposób oszczędzają firmie niepotrzebnych wydatków i nie narażają jej dobrego imienia na szwank, wykrywając zawczasu słabe punkty programów. W tym rozdziale dokonamy przeglądu różnych technik atakowania aplikacji, wykorzystując zdobyte wcześniej wiadomości z zakresu reinżynierii oprogramowania.

Przepełnienie bufora

Aby dokonać ataku przepełnienia bufora, potrzebna jest bardzo dobra znajomość asemblera, C++ i atakowanego systemu operacyjnego. W tym podrozdziale opiszemy ataki przepełnienia bufora, prześledzimy ich ewolucję i zademonstrujemy przykład takiego ataku.

Przepełnienie bufora (ang. *buffer overflow*) polega na podaniu programowi większej liczby danych, niż to przewidział twórca programu. Nadmiarowe dane przekraczają obszar pamięci, który został przeznaczony dla danych, a tym samym wkraczają w obszar pamięci, który był przeznaczony na instrukcje programu. W idealnej wersji tego ataku nadmiarowe dane są nowymi instrukcjami, które umożliwiają napastnikowi sterowanie pracą procesora.

Ataki przepełnienia bufora nie są niczym nowym, już w 1988 roku robak Morris wykorzystywał tę technikę. Samo zagrożenie dla bezpieczeństwa systemów komputerowych, spowodowane przepełnieniem bufora, znane jest od lat 60. ubiegłego wieku.

Przykład przepełnienia bufora

Możliwość przepełniania bufora bierze się z wrodzonej słabości języka C++. Problem, który został odziedziczony z języka C i który dotyczy również innych języków programowania, takich jak Fortran, polega na tym, że C++ nie kontroluje automatycznie długości

podawanych programowi danych. Aby to lepiej zrozumieć, posłużmy się przykładem kodu przedstawiającego, jak funkcje C/C++ zwracają dane programowi głównemu:

```
// lunch.cpp : Przepełnienie bufora stomach

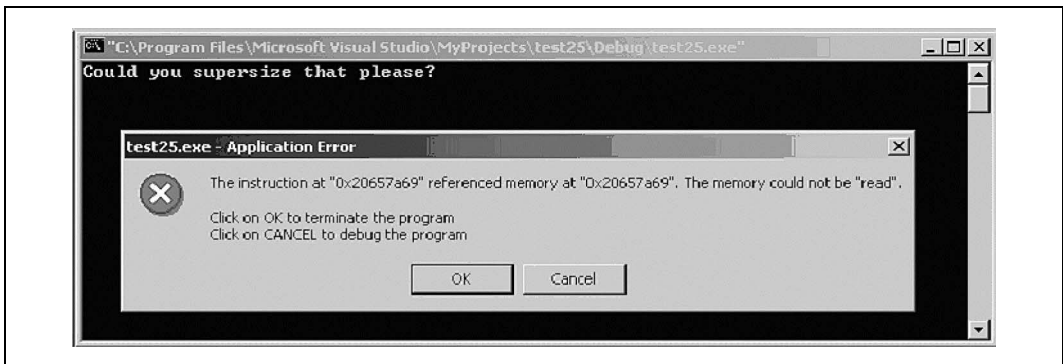
#include <stdafx.h>
#include <stdio.h>
#include <string.h>

void bigmac(char *p);

int main(int argc, char *argv[])
{
    bigmac("Could you supersize that please?"); //przepełnienie spowoduje
                                                wpisanie >9 znaków
    return 0;
}

void bigmac(char *p)
{
    char stomach[10]; //ograniczamy rozmiar bufora do 10 znaków
    strcpy(stomach, p);
    printf(stomach);
}
```

Aby sprawdzić działanie programu, należy skompilować go za pomocą kompilatora C++. Chociaż kompilacja przebiegnie bez przeszkód, wykonywanie uruchomionego programu załamie się, co przedstawiono na rysunku 5.1.



Rysunek 5.1. Załamanie programu na skutek przepełnienia bufora

Co się stało? Po uruchomieniu programu, funkcji `bigmac` przekazywany jest długi łańcuch znaków „Could you supersize that please?”. Niestety funkcja `strcpy()` nie sprawdza długości kopiowanych łańcuchów. Jest to bardzo niebezpieczne, gdyż przekazanie funkcji łańcucha o długości większej niż dziewięć znaków powoduje przepełnienie bufora.

Funkcja `strcpy()`, podobnie jak kilka innych funkcji języka C++, ma istotną wadę, która powoduje, że nadmiarowe znaki są wpisywane do obszaru pamięci leżącego poza zmienną. Z reguły powoduje to załamanie się programu. W tym konkretnym przypadku załamanie programu spowodowała próba czytania poza granicą statycznego łańcucha.

W najgorszym przypadku tego rodzaju przepełnienie umożliwia wykonanie w zaatakowanym systemie dowolnego kodu, o czym powiemy w dalszej części rozdziału.

Wyjaśnienie pojęcia bufora

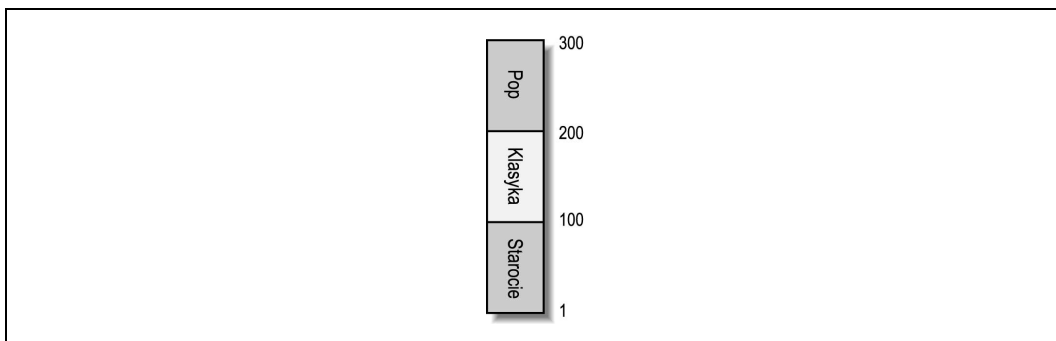
Podatność na przepełnienie bufora jest najczęściej wykorzystywaną luką w systemie bezpieczeństwa. Aby zrozumieć, jak kraker wykorzystuje przepełnienie bufora do wniknięcia lub załamania pracy komputera, należy zrozumieć, czym jest bufor.



W podrozdziale wyjaśniamy, co to jest bufor. Zaawansowani czytelnicy mogą przejść do następnego podrozdziału „Uderzenie w stos”.

Program komputerowy składa się kodu, który korzysta ze zmiennych zapisanych w różnych miejscach pamięci. Podczas wykonywania się programu zmiennym przydzielana jest pamięć, a jej ilość zależy od rodzaju informacji, które ma przechowywać zmienna. Na przykład dane typu Short Integer zajmują niewiele miejsca w pamięci RAM, natomiast dane Long Integer zajmują go więcej. Istnieją różne rodzaje zmiennych, a każdy rodzaj może potrzebować innej ilości pamięci. Przestrzeń zarezerwowana w pamięci dla zmiennej jest używana do przechowywania informacji niezbędnych do wykonywania się programu. Program przechowuje w tym obszarze pamięci zawartość zmiennej i w razie potrzeby pobiera ją stamtąd. Ta wirtualna przestrzeń nazywana jest buforem.

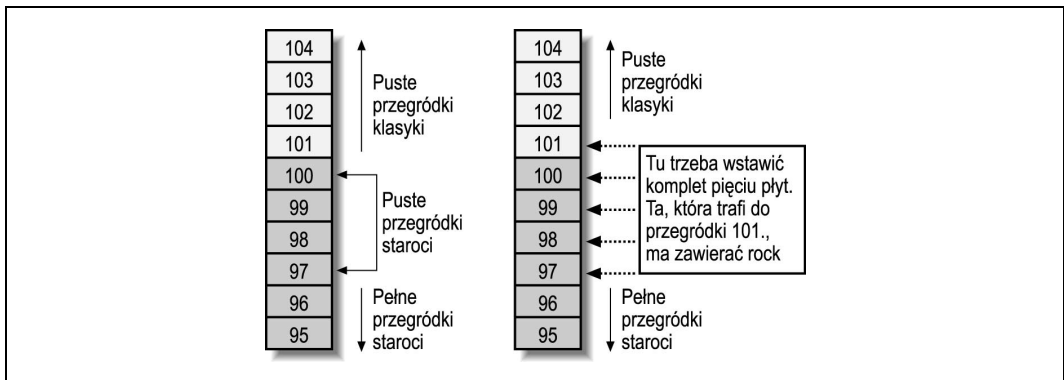
Dobłą analogią bufora jest kolekcja płyt CD przechowywana w stojaku z — dajmy na to — 300 przegródkami. Pamięć komputera jest podobna do takiego stojaka, choć oczywiście liczba przegródek, w których komputer umieszcza informacje, jest liczona w milionach. Płyty z naszego zbioru należą do trzech kategorii — staroci, klasyki i popu (rysunek 5.2). Logicznie podzielmy stojak na trzy równe części. Dolne 100 przegródek stojaka CD będzie mieścić starocie, środkowe — klasykę, a górne — pop. Każda przegródka stojaka otrzyma własny numer. Na podstawie numeru przegródki wiadomo, gdzie zaczynają się i kończą płyty każdej kategorii.



Rysunek 5.2. Podzielony na części stojak jest podobny do bufora

Pamięć komputera wygląda bardzo podobnie. Gdy program jest ładowany, przydziela wszystkim zmiennym, których ma używać, fragmenty pamięci. Jedna zmienna programu może zajmować kilka przegródek w pamięci. Taka sama sytuacja ma miejsce również w przypadku płyt CD — żeby umieścić w stojaku czteropłytkowy album z utworami Bacha, należy użyć czterech kolejnych przegródek. Ta część pamięci nazywa się buforem. Mówiąc prosto, bufor to fragment pamięci komputera, który program zarezerwował na zawartość zmiennej, aby w razie potrzeby móc ją stamtąd pobierać.

Mając już ogólne pojęcie o buforze, wyjaśnijmy, na czym polega jego przepełnienie. Wyjaśnieniu towarzyszy rysunek 5.3, na którym widać przegródki stojaka przeznaczone na starocie (1 – 100) oraz klasykę (101 – 200). Załóżmy teraz, że jest to kolekcja płyt kolegi, a ponieważ nie lubimy ani staroci, ani klasyki, ani popu, chcemy zmusić kolegę, żeby zagrał naszą płytę CD z muzyką rockową.



Rysunek 5.3. Przykład przepełnienia bufora

Co wiemy o organizacji płyt na stojaku CD kolegi? Wiemy, że przegródki stojaka są podzielone następująco: 1 – 100, 101 – 200, 201 – 300. Sekcja staroci (1 – 100) jest już prawie wypełniona i ma tylko cztery wolne przegródki (97 – 100), natomiast sekcja klasyki pozostaje zupełnie pusta. Możemy teraz dać koledze zestaw pięciu płyt Barry’ego Manilowa (dla potrzeb przykładu założmy, że to kategoria staroci), w którym w miejscu piątej płyty przemyciliśmy płytę rockową. Jeśli kolega, wkładając płyty do stojaka, nie zwraca uwagi na numery przegródek, naszą płytę rockową umieści w przegródce 101. Teraz wystarczy poprosić kolegę, by zagrał coś z kolekcji płyt z muzyką klasyczną. Kolega sprawdzi numery przegródek, stwierdzi, że w sekcji z muzyką klasyczną znajduje się jedna płyta, wyjmie ją i umieści w odtwarzaczu. Ku swojemu zdziwieniu zamiast muzyki Beethovena usłyszy muzykę rockową.

W bardzo podobny sposób haker dokonuje ataku przepełnienia bufora w komputerze. Najpierw musi znaleźć działający program, który jest podatny na przepełnienie bufora. Nawet gdy działanie hakera nie zakończy się uruchomieniem szkodliwego kodu, z pewnością doprowadzi do załamania programu. Haker musi też znać dokładny rozmiar bufora, który będzie chciał przepełnić. W przypadku stojaka z płytami CD wystarczyło podarować przyjacielowi pięć płyt CD, z których jedna przepełniła segment staroci. W przypadku komputerów czasem bywa to równie proste.

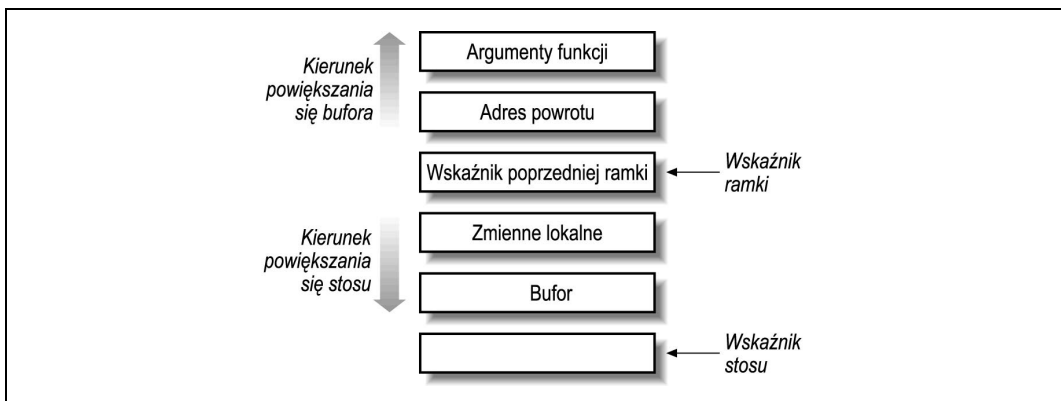
Dobrze napisany program nie pozwoli sobie na przepełnienie bufora — to tak, jakby użyć trzech różnych stojaków na płyty CD po sto przegródek każdy, a nie jednego stojaka o trzystu przegródkach. Gdyby kolega z przykładu miał trzy różne stojaki, zauważyłby, że kolekcja staroci zawiera o jedną płytę za dużo, i podjąłby odpowiednie działania, które prawdopodobnie doprowadziłyby do odkrycia przemyczonej w zestawie płyty z muzyką rockową.

Drugim etapem ataku przepełnienia bufora jest wykonanie dostarczonego kodu (*ładunku*). Ładunkiem jest zwykle polecenie umożliwiające zdalny dostęp do zaatakowanego komputera lub inne polecenie ułatwiające hakerowi przejęcie komputera. Na przykład serwer Internet Information Server firmy Microsoft miał słaby punkt umożliwiający napastnikowi utworzenie kopii dowolnego pliku i umieszczenie jej w części dostępnej przez WWW. Plikiem tym mogło być wszystko, co umożliwiało zdalny dostęp — począwszy od pliku haseł, a skończywszy na pliku programu.

Przeprowadzenie skutecznego ataku przepełnienia bufora jest trudne. Jednak nawet wtedy, gdy atak przepełnienia bufora nie powiedzie się do końca, najczęściej i tak spowoduje problemy w atakowanym komputerze. Zwykle polegają one na załamaniu się działania zaatakowanego programu lub nawet całego systemu. Program, do którego należała zajęta w wyniku ataku pamięć, nie sprawdza, czy jej zawartość nie zmieniła się, dlatego użyje jej, zakładając, że jest to ta sama zawartość, z której korzystał ostatnio. Gdy na przykład zechce odczytać liczbę używaną do dalszych obliczeń, a zamiast niej otrzyma wyraz „Bob”, nie będzie wiedział, co z nim począć.

Uderzenie w stos

W tym podrozdziale opiszemy mechanizm typowego przepełnienia bufora. Na rysunku 5.4 przedstawiono widok stosu po wywołaniu funkcji. Wskaźnik stosu wskazuje na wierzchołek stosu znajdujący się u dołu rysunku.



Rysunek 5.4. Wygląd stosu po wywołaniu funkcji

C++ umieszcza na stosie następujące dane — zmienne lokalne, wskaźnik poprzedniej ramki, adres powrotu oraz argumenty funkcji. Dane te nazywane są *ramką* funkcji i reprezentują stan funkcji. Wskaźnik ramki lokalizuje bieżącą ramkę, a wskaźnik poprzedniej ramki przechowuje wskaźnik ramki funkcji, która wywołała daną funkcję.

Gdy napastnik przepełni bufor znajdujący się na stosie (podając więcej danych, niż może on przyjąć), bufor powiększy się w kierunku leżącego na stosie adresu powrotu. Celem hakera jest zmiana tego adresu. Gdy działanie funkcji kończy się, adres powrotu pobierany jest ze stosu, po czym następuje skok pod ten adres. Zmieniając adres powrotu, haker może przejąć kontrolę nad procesorem. Jeżeli zmieniony adres powrotu będzie wskazywał na szkodliwy kod umieszczony w przepełnionym buforze, kod zostanie wykonany z uprawnieniami zaatakowanego programu.

Przepełnienie sterty

Z powodu szerokiego nagłośnienia problemu oraz w wyniku stosowania technik prewencyjnych, omawianych w dalszej części rozdziału, ataki przepełnienia bufora stają się coraz rzadsze. W efekcie mamy coraz częściej do czynienia z atakami przepełnienia sterty.

Sterta (ang. *heap*) nazywamy obszar pamięci dynamicznie przydzielonej aplikacji w celu przechowywania w niej zmiennych. W ataku *przepełnienia sterty* haker usiłuje zmienić zawartość takich zmiennych, jak hasła, nazwy plików oraz identyfikatory UID znajdujące się na stercie.

Jaka jest różnica między przepełnieniem bufora a przepełnieniem sterty? W przypadku przepełnienia bufora zmiana zawartości adresu powrotu znajdującego się na stosie oznacza próbę wykonania określonych instrukcji maszynowych. W przypadku przepełnienia sterty zmiana zawartości zmiennych dynamicznych oznacza próbę podniesienia poziomu uprawnień w systemie. Programy stosujące przepełnienie sterty wykorzystują błędy formatów oraz atakują funkcje `malloc( )` i `free( )`.

Badacze zaczęli wyróżniać osobną klasę przepełnień nazywaną *błędami formatów* (ang. *format bugs*). Słabym punktem języka C jest istnienie specyfikatora formatu `%n`, który każe funkcji `printf` zapisać do podanego funkcji argumentu typu `int *` liczbę sformatowanych do tej pory bajtów. Mechanizm ten można wykorzystać, gdy program nie filtruje otrzymywanych od użytkownika danych, używanych jako pierwszy argument funkcji `printf`. Mechanizm `varargs` języka C++ umożliwia funkcjom (np. funkcji `printf`) używanie zmiennej liczby argumentów. Funkcje pobierają ze stosu tyle argumentów (i takiego typu), ile określa pierwszy argument funkcji `printf`. Problemowi temu można zapobiec, używając zamiast funkcji `printf(buf)` funkcji `printf("%s", buf)`.

Zapobieganie przepełnieniom bufora

Najlepszym sposobem, by uniemożliwić ataki przepełnienia bufora, jest przestrzeganie przez programistę właściwych praktyk programowania. Należą do nich:

- sprawdzanie wielkości danych przed umieszczeniem ich w buforze,
- użycie funkcji ograniczających liczbę i (lub) format wprowadzanych znaków,
- unikanie niebezpiecznych funkcji języka C, takich jak `scanf( )`, `strcpy( )`, `strcat( )`, `getwd( )`, `gets( )`, `strcmp( )`, `sprintf( )`.

Jak w praktyce zapobiegać przepełnianiu bufora? Programiści używają funkcji `strcpy( )` kopiującej łańcuch źródłowy do bufora docelowego. Niestety bufor docelowy może nie być na tyle pojemny, by pomieścić łańcuch źródłowy. Gdy użytkownik wprowadzi bardzo długi łańcuch źródłowy, spowoduje przepełnienie bufora docelowego.

Aby zapobiec tego rodzaju błędom, należy przed skopiowaniem każdego łańcucha źródłowego sprawdzać jego długość. Prostszy sposób może być użycie funkcji `strncpy( )`. Funkcja ta działa podobnie do funkcji `strcpy( )`, a różni się od niej tym, że kopiuje tylko pierwszych *n* bajtów łańcucha źródłowego, co umożliwia ochronę przed przepełnianiem bufora.

Zautomatyzowane badanie kodu

Nie urodził się jeszcze programista, który nie popełnia błędów. Dlatego teraz zajmiemy się narzędziami, których zadaniem jest badanie podatności kodu na przepełnienie bufora.

Jeszcze nie tak dawno brakowało skutecznych narzędzi do automatycznego badania kodu źródłowego pod kątem podatności na przepełnienie bufora. Uwzględnienie wszystkich możliwych przyczyn tego rodzaju błędów w programach liczących tysiące wierszy kodu jest bowiem niezwykle trudne.

Jednym z komercyjnych programów służących do tego celu jest PolySpace (<http://www.polyspace.com>), zawierający narzędzie do wykrywania w czasie kompilacji możliwości przepełnienia bufora w programach napisanych w ANSI C. I choć moduł Viewer może działać w systemie Windows, to zasadniczy moduł Verifier wymaga użycia Linuksa. Programiści, którzy piszą tylko dla systemu Windows, ale chcą posłużyć się programem PolySpace, będą musieli przemóc się i zainstalować gdzieś system Linux. Wyniki działania programu można analizować już w systemie Windows. Jeżeli ktoś jeszcze nie używa Linuksa, powinien zrobić to niezwłocznie — prawdziwy specjalista ds. bezpieczeństwa powinien z łatwością poruszać się w obu środowiskach. Jednak mając na uwadze zatawardziały linuksofobów, firma PolySpace zaczęła przenosić motor programu Verifier do systemu Windows.

Dodatki do kompilatora

Dla Linuksa dostępne są dodatki do kompilatora C++ oraz biblioteki, które sprawdzają w czasie wykonywania się programu wielkość wprowadzanych danych. Przykładem tego może być StackGuard (<http://immunix.org>), który wykrywa ataki uderzenia w stos, chroniąc przed zmianą adres powrotu funkcji. Podczas wywołania funkcji umieszcza on

obok adresu powrotu bajty „przynęty”, natomiast podczas kończenia pracy funkcji sprawdza, czy przynęta została zmieniona. Jeżeli tak się stało, umieszcza zapis w dzienniku zdarzeń systemu i kończy działanie programu.

StackGuard jest zaimplementowany w postaci niewielkiej łatki na generator kodu kompilatora gcc, umieszczanej w procedurach `function_prolog()` i `function_epilog()`. Używa funkcji `function_prolog()` do ułożenia „przynęty” na stosie, a gdy funkcja kończy działanie, za pomocą funkcji `function_epilog()` bada stan „przynęty”. W ten sposób może wykryć każdą próbę uszkodzenia adresu powrotu jeszcze przed zakończeniem działania funkcji.

Innym przydatnym programem firmy `immunix.org` jest FormatGuard, który zabezpiecza przed błędami formatów. Wykorzystuje on cechę języka C++ polegającą na odróżnianiu makr o identycznych nazwach, ale różniących się liczbą argumentów. FormatGuard zawiera definicje makr funkcji `printf` dla od jednego do stu argumentów. Każde z tych makr wywołuje bezpieczną osłonę, która zlicza w łańcuchu formatu liczbę znaków „%” i odmawia wywołania, gdy liczba argumentów nie odpowiada liczbie specyfikatorów %.

Żeby chronić program za pomocą FormatGuard, należy skompilować go, wykorzystując nagłówki glibc pochodzące z FormatGuard. FormatGuard jest osłoną funkcji `libc`: `syslog()`, `printf()`, `fprintf()`, `sprintf()`, `snprintf()`.

Inne sposoby ochrony

Inny sposób zapobiegający przepełnieniu bufora polega na uniemożliwianiu wykonywania kodu w obszarze pamięci stosu i danych. Nie jest to co prawda kompletne rozwiązanie i napastnik wciąż może wykonać skok w nieprzewidziane miejsce, mimo to utrudnia znacznie wykonanie kodu. Rozwiązanie to dostępne jest w postaci łatki na system Linux.

Dodatkową linię obrony mogą stanowić narzędzia kontrolujące rozmiar danych. W przeciwieństwie do C++, Perl i Java posiadają wbudowane mechanizmy kontroli długości danych, które zwalniają programistę od tworzenia własnych mechanizmów zabezpieczających. Wykorzystując narzędzia dostępne w różnych systemach operacyjnych, w możliwość taką można wyposażyć również C++. Do narzędzi tego typu należą BOWall, Boundschecker firmy Compuware czy Purify firmy Rational.

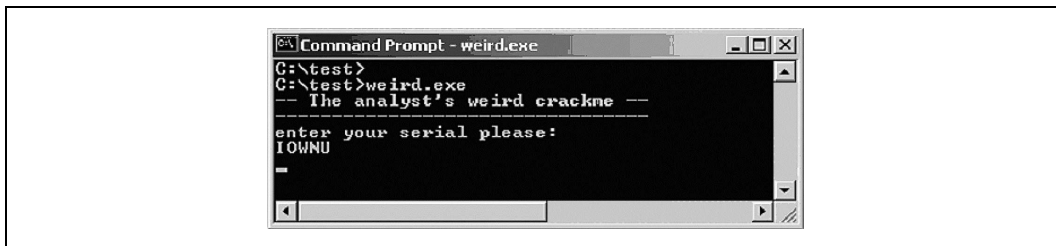
Przepełnienie bufora w praktyce

Po zapoznaniu się z zagadnieniami związanymi z przepełnieniem bufora, pora przejść do przykładu umożliwiającego sprawdzenie zdobytej wiedzy na odpowiednim programie testowym.



W przykładzie posłużymy się programem typu crackme o nazwie *weird.exe*. Program można pobrać ze strony <http://www.securitywarrior.com>.

Program stworzył Analyst, a prezentowany tu przykład opublikował +Tsehpb, dzięki uprzejmości którego możemy go przedstawić. Uruchomiony w wierszu poleceń, program prosi o podanie numeru seryjnego (rysunek 5.5), którego oczywiście nie znamy, ale możemy go zgadnąć. Kiedy to się uda, program wyświetli komunikat z gratulacjami.



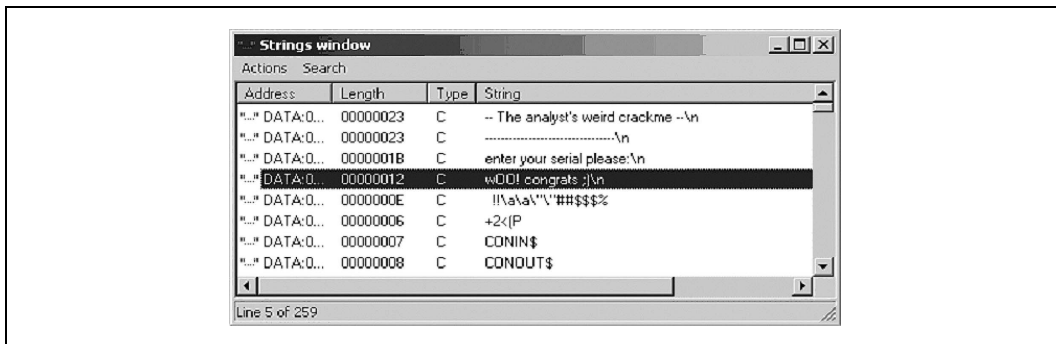
Rysunek 5.5. Program *weird.exe* służący do testowania przepełnienia bufora

Zacznijmy od wpisania numeru seryjnego „IOWNU”. Ponieważ jest on niepoprawny, nie spowoduje żadnej reakcji programu, a ponowne wciśnięcie klawisza *Enter* zakończy pracę programu. Po kilku dalszych próbach dojdziemy do wniosku, że istnieje lepszy sposób odkrycia poprawnego numeru seryjnego. Można co prawda napisać program, który będzie próbował odgadnąć numer seryjny metodą „brute-force”, ale jest to rozwiązanie nieeleganckie. Nadszedł czas, aby skorzystać z narzędzi reinyżynierii oprogramowania systemu Windows. Jedyną regułą tej zabawy jest zakaz zmieniania instrukcji badanego programu.

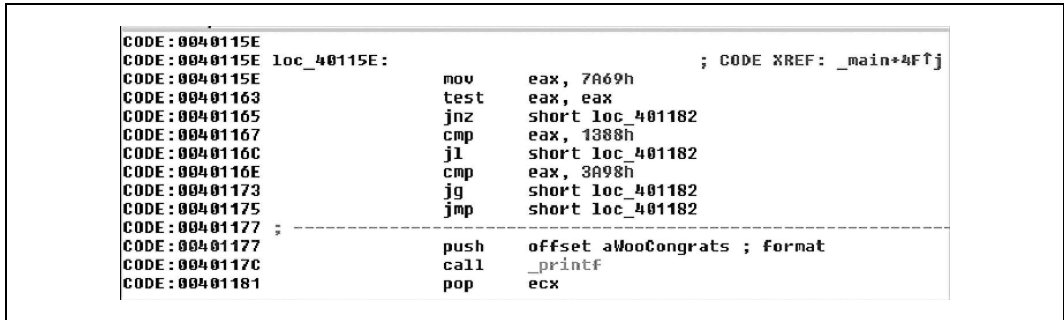
Poprawny numer seryjny znajdziemy, wykorzystując techniki reinyżynierii oprogramowania poznane w poprzednich rozdziałach. Potrzebne do tego będą:

- znajomość asemblera x86,
- deassembler taki, jak IDA Pro lub W32DASM,
- konwerter kodów szesnastkowych na znaki ASCII.

Zacynamy od uruchomienia programu IDA Pro i otwarcia pliku *weird.exe*. Przechodzimy od razu do okna *Strings*, w którym szukamy łańcucha znaków zawierającego gratulacje (rysunek 5.6). Po dwukrotnym kliknięciu łańcucha na ekranie pojawia się kod, któremu należy się przyjrzeć (rysunek 5.7).



Rysunek 5.6. Łańcuchy znaków znajdujące się w programie *weird.exe*



Rysunek 5.7. Badany fragment kodu widziany w programie IDA Pro

Nie ma żadnych żelaznych i dających szybkie wyniki reguł łamania programów. Reinżynieria oprogramowania jest bardziej sztuką niż wiedzą i często w takim samym stopniu, co na wiedzy i doświadczeniu, opiera się na szczęściu i intuicji. W tym przypadku rozpoczęliśmy od kodu używającego łańcucha znaków z gratulacjami, gdyż wygląda on obiecująco.

Oto kod programu:

```

CODE:00401108      push    ebp
CODE:00401109      mov     ebp, esp
CODE:0040110B      add     esp, 0FFFFFFB4h ; znak
CODE:0040110E      push    offset aTheAnalystSWei ; __va_args
CODE:00401113      call    _printf          ; pokaż tekst
CODE:00401118      pop     ecx
CODE:00401119      push    offset asc_40C097 ; __va_args
CODE:0040111E      call    _printf          ; jak wyżej
CODE:00401123      pop     ecx
CODE:00401124      push    offset aEnterYourSeria ; __va_args
CODE:00401129      call    _printf          ; i tu też
CODE:0040112E      pop     ecx
CODE:0040112F      lea     eax, [ebp+s]      ; bufor
CODE:00401132      push    eax              ; s
CODE:00401133      call    _gets            ; pobiera podany numer seryjny
CODE:00401138      pop     ecx
CODE:00401139      nop
CODE:0040113A      lea     edx, [ebp+s]
CODE:0040113D      push    edx              ; s
CODE:0040113E      call    _strlen          ; pobiera jego długość
CODE:00401143      pop     ecx
CODE:00401144      mov     edx, eax
CODE:00401146      cmp     edx, 19h         ; czy jest krótszy niż 25?
CODE:00401149      jl      short loc_401182 ; tak
CODE:0040114B      cmp     edx, 78h         ; czy jest dłuższy niż 120?
CODE:0040114E      jg      short loc_401182 ; tak
CODE:00401150      mov     eax, 1           ; eax = 1, inicjacja pętli
CODE:00401155      cmp     edx, eax         ; czy to już wszystkie znaki?
CODE:00401157      jl      short loc_40115E ; nie, dlatego wykonuje skok
CODE:00401159      ; -----
CODE:00401159      loc_401159:             ; CODE XREF: _main+54j
CODE:00401159      inc     eax              ; eax = eax + 1
CODE:0040115A      cmp     edx, eax         ; czy to już wszystkie znaki?
CODE:0040115C      jge     short loc_401159 ; nie, kontynuowanie pętli

```

```

CODE:0040115E
CODE:0040115E loc_40115E:                ; CODE XREF: _main+4Fj
CODE:0040115E     mov     eax, 7A69h      ; eax = 31337
CODE:00401163     test    eax, eax
CODE:00401165     jnz     short loc_401182      ; wyjście
CODE:00401167     cmp     eax, 1388h
CODE:0040116C     jnl     short loc_401182      ; wyjście
CODE:0040116E     cmp     eax, 3A98h
CODE:00401173     jg      short loc_401182      ; wyjście
CODE:00401175     jmp     short loc_401182      ; wyjście
CODE:00401177 ; -----
CODE:00401177     push    offset aWooCongrats ; _va_args
                        ; komunikat o powodzeniu
CODE:0040117C     call   _printf
CODE:00401181     pop     ecx
CODE:00401182
CODE:00401182 loc_401182:                ; CODE XREF: _main+41j
CODE:00401182                        ; _main+46j ...
CODE:00401182     call   _getch ; czekanie na naciśnięcie klawisza
CODE:00401187     xor     eax, eax
CODE:00401189     mov     esp, ebp
CODE:0040118B     pop     ebp
CODE:0040118C     retn

```

W kodzie odkrywamy podstęp. Wygląda na to, że w ogóle nie istnieje możliwość zobaczenia komunikatu z gratulacjami! Po przejrzeniu kodu stwierdzamy, że nie ma w nim odwołania do tej części kodu, która odpowiada za wyświetlenie gratulacji na ekranie, a są tylko skoki prowadzące do zakończenia działania programu. To dziwne, ale pewnie dlatego program nazywa się *weird.exe*¹.

Wydaje się, że jedynym sposobem, żeby zobaczyć komunikat z gratulacjami, jest wykonanie wyświetlającego go kodu przez przepełnienie bufora programu. Mówiąc inaczej, sami musimy stworzyć taki numer seryjny, który przepełniając bufor programu, zmusi go do wykonania określonego kodu. Żeby wykonać ten kod, musimy na stosie umieścić numer seryjny przygotowany w bardzo precyzyjny sposób.

Zacząć należy od sprawdzenia położenia i wielkości bufora:

```

CODE:0040112E     pop     ecx
CODE:0040112F     lea     eax, [ebp+s] ; bufor

CODE:00401132     push    eax ; s
CODE:00401133     call   _gets ; pobiera podany numer seryjny

CODE:00401138     pop     ecx
CODE:00401139     nop
CODE:0040113A     lea     edx, [ebp+s]
CODE:0040113D     push    edx ; s

```

Widzimy, że zawartość `eax` trafia na stos bezpośrednio przed wywołaniem funkcji `gets ( )`.

¹ Dziwak — przyp. tłum.

I znów nastąpiło załamanie programu, któremu tym razem towarzyszy komunikat:

```
Instruction at the address 41414141h uses the memory address at 41414141h.
The memory
cannot be "read".2
```

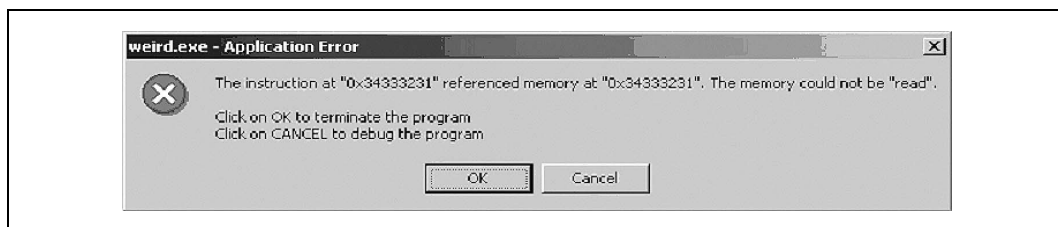
Z tego wniosek, że program próbuje wykonać instrukcję znajdującą się pod adresem 41414141h. A gdyby tak zastąpić adres powrotu czymś innym niż 41414141h? Na przykład adresem kodu pokazującego komunikat z gratulacjami?

```
CODE:00401177                push    offset aWooCongrats
                           ; _va_args ; dobry chłopiec
CODE:0040117C                call     _printf
```

Jeżeli udałooby się umieścić w miejsce adresu powrotu adres 401177, program wykonałby kod wyświetlający komunikat z gratulacjami. Zanim to jednak zrobimy, użyjmy jeszcze takiego numeru seryjnego:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA1234
```

Jak widać na rysunku 5.8, wykonywanie programu załamało się pod adresem 34333231.



Rysunek 5.8. Załamanie wykonywania się programu

Uświadamia nam to, że musimy zmienić porządek bajtów w dostarczonym programowi ładunku. Dlaczego? Ponieważ adres, pod którym nastąpiło załamanie wykonywania się programu, jest odwróceniem porządku kodów ASCII ostatnich czterech podanych znaków.

Oderwijmy się na chwilę od przykładu i wyjaśnijmy przyczynę tego stanu rzeczy. Odwrotny porządek jest potrzebny w przypadku procesorów x86, gdyż używają one zapisu typu little-endian. Termin „endian” wywodzi się z *Podróży Guliwera* Jonathana Swifta. Jest to satyryczna opowieść o mieszkańcach dwóch miast, którzy obierali ugotowane na twardo jajka, stukając je w różne końce. Mieszkańcy jednego miasta stukali jajka w ostrzejszy koniec (ang. *big end*), a drugiego — w bardziej zaokrąglony (ang. *little end*). Ta różnica w zachowaniu mieszkańców doprowadziła z czasem do wojny.

W przypadku procesorów określenia „big-endian” i „little-endian” dotyczą porządku bajtów liczb wielobajtowych. W formacie big-endian najbardziej znaczący bajt³ liczby

² Instrukcja pod adresem 41414141h korzysta z pamięci o adresie 41414141h. Pamięć nie daje się czytać — przyp. tłum.

³ Czyli bajt, który w zapisie liczby na kartce znajduje się na lewym końcu liczby — przyp. tłum.

zapisywany jest pod najmniejszym adresem pamięci, gdy w formacie little-endian, pod najniższym adresem zapisywany jest najmniej znaczący bajt. Operując na zawartości adresów, należy znać porządek, w jakim dany procesor zapisuje i odczytuje bajty z pamięci.

Wróćmy do przykładu. Wiemy, że kodami szesnastkowymi znaków ASCII 1-2-3-4 są liczby 31-32-33-34. Jeżeli zmienimy porządek 1-2-3-4 na 4-3-2-1 otrzymamy liczby 34-33-32-31, które dają adres 34333231, pod którym załamało się wykonywanie programu. Dlatego żeby skutecznie złamać program, musimy zamienić porządek bajtów podawanego programowi adresu. Mówiąc inaczej, aby dokonać skoku pod adres 401177, musimy na stosie umieścić liczbę 771140. Liczbie 771140 odpowiada ciąg znaków ASCII w^Q@ (gdzie ^Q oznacza *Ctrl+Q*).

Spróbujmy podać go teraz programowi:

```
-- The analyst's weird crackme --
-----
enter your serial please:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAw^Q@
```

Po naciśnięciu klawisza *Enter* na ekranie wyświetlą się gratulacje (rysunek 5.9):

```
wOO! congrats ;)
```



Rysunek 5.9. Gratulacje

Udało nam się z powodzeniem wykorzystać możliwość przepełnienia bufora do wykonania instrukcji umieszczonych na stosie. W tym przypadku na stosie umieściliśmy adres, który doprowadził do kodu, niedostępnego w żaden inny sposób.

Po złamaniu programu można zapoznać się z jego kodem źródłowym:

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include <iostream.h>

int main( ){
    int i,len,temp;

    unsigned char name[75];
```

```
unsigned long check=0;

printf("-- The analyst's weird crackme --\n");
printf("-----\n");
printf("enter your serial please:\n");

gets(name);
asm{ nop};

len=strlen(name);

//cout << len;
if (len < 25) goto theend;
if (len > 120 ) goto theend;
for (i=1; i <= len ; i++)
{
    temp += name[i] ;
}

if (temp = 31337) goto theend;
if (temp < 5000) goto theend;
if (temp > 15000) goto theend;

goto theend;

    printf("wOO! congrats ;)\n");

theend:

getch( );
return 0;
```

Źródła dodatkowych informacji

- *Building Secure Software: How to Avoid Security Problems the Right Way*, John Viega i Gary McGraw, Addison-Wesley Professional, 2001.
- The Analyst's weird crackme. Publikacja +Tsehp, 2001.
- *Smashing the Stack for Fun and Profit*, Aleph One, Phrack numer 49, 14 listopada 1996 (<http://www.phrack.org>).
- „Endian Issues”, William Stallings, Byte.com, wrzesień 1995.