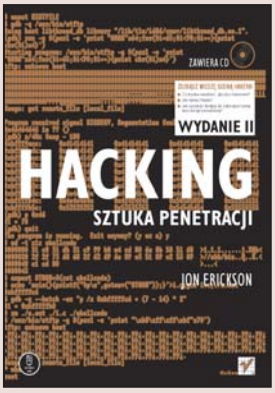
Hacking. Sztuka penetracji. Wydanie II



Autor: Jon Erickson
Tłumaczenie: Marcin Rogóż
ISBN: 978-83-246-1802-6
Tytuł oryginału: Hacking:
The Art of Exploitation, 2nd Edition
Format: 170x230, stron: 520

Zdobądź wiedzę godną hakera!

- Co trzeba wiedzieć, aby być hakerem?
- Jak łamać hasła?
- Jak uzyskać dostęp do zabezpieczonej sieci bezprzewodowej?

Słowo haker kojarzy nam się z komputerowym mistrzem manipulacji Kevinem Mitnickiem. Pojęcie to jednak ewoluowało od czasu jego spektakularnych akcji. Zatem kim jest dziś haker? Wbrew obiegowej opinii, większość hakerów nie wykorzystuje swej wiedzy do niecnych celów. Dzięki swej wiedzy i dociekliwości przyczyniają się do rozwoju bezpieczeństwa sieci i programów. Nikt bowiem nie potrafi tak jak oni zgłębiać sposoby działania zaawansowanego oprogramowania i sprzętu i tropić luki pozwalające na atak lub wyciek danych oraz przewidywać możliwe problemy w ich działaniu.

Jon Erickson w książce Hacking. Sztuka penetracji. Wydanie II omawia te zagadnienia ze świata informatyki, które nie mogą być obce hakerowi. Dzięki tej książce poznasz m.in. podstawy języka C oraz dowiesz się, jak wykorzystać jego słabości oraz potknięcia programistów piszących w tym języku. Zapoznasz się z podstawami funkcjonowania sieci i zdobędziesz wiedzę o modelu OSI, a także nauczysz się podsłuchiwać transmitowane w sieci dane, skanować porty i łamać hasła.

- Programowanie w języku C
- Model OSI
- Podsłuchiwanie sieci
- Skanowanie portów
- Sposoby łamania haseł
- Szyfrowanie danych i połączeń
- Sposoby atakowania sieci bezprzewodowych

Oto elementarz prawdziwego hakera!

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

SPIS TREŚCI

0x200

PROGRAMOWANIE

Haker to termin używany do określenia zarówno piszących kod, jak i tych, którzy wykorzystują znajdujące się w nim błędy. Choć obie grupy hakerów mają różne cele, to i jedni, i drudzy stosują podobne techniki rozwiązywania problemów. A ze względu na fakt, że umiejętność programowania pomaga tym, którzy wykorzystują błędy w kodzie, zaś zrozumienie metod zastosowania pomaga tym, którzy programują, wielu hakerów robi obie rzeczy jednocześnie. Interesujące rozwiązania istnieją i w zakresie technik pisania eleganckiego kodu, i technik służących do wykorzystywania słabości programów. Hakerstwo to w rzeczywistości akt znajdowania pomysłowych i nieintuicyjnych rozwiązań problemów.

Metody stosowane w narzędziach wykorzystujących słabości programów zwykle są związane z zastosowaniem reguł funkcjonowania komputera w sposób, którego nigdy nie brano pod uwagę — pozornie magiczne rezultaty osiąga się zwykle po skupieniu się na omijaniu zaimplementowanych zabezpieczeń. Metody używane podczas pisania programów są podobne, bo również wykorzystują reguły funkcjonowania komputera w nowy i pomysłowy sposób, jednak w tym przypadku końcowym celem jest osiągnięcie najbardziej wydajnego lub najkrótszego kodu źródłowego. Istnieje nieskończenie wiele programów, które można napisać w celu wykonania dowolnego zadania, jednak większość z istniejących rozwiązań jest niepotrzebnie obszerna, złożona i niedopracowana. Pozostała niewielka ich liczba to programy nieduże, wydajne i estetyczne. Ta właściwość programów nosi nazwę *elegancji*, zaś przemyślane i pomysłowe rozwiązania prowadzące do takiej wydajności nazywane są *sztuczkami* (ang. *hacks*). Hakerzy stojący po obu stronach metodyki programowania doceniają zarówno piękno eleganckiego kodu, jak i geniusz pomysłowych sztuczek.

W świecie biznesu mniejszą wagę przykłada się do przemyślanych metod programowania i eleganckiego kodu, a większą do tworzenia funkcjonalnego kodu w jak najkrótszym czasie i jak najtaniej. Ze względu na gwałtowny wzrost mocy obliczeniowej oraz dostępności pamięci poświęcenie dodatkowych pięciu godzin na opracowanie nieco szybszego i mniej wymagającego pod względem pamięci fragmentu kodu po prostu nie opłaca się, gdy mamy do czynienia z nowoczesnymi komputerami dysponującymi gigahercowymi cyklami procesora i gigabajtami pamięci. Podczas gdy optymalizacje czasu i wykorzystania pamięci pozostają niezauważone przez większość (z wyjątkiem wyrafinowanych) użytkowników, nowa funkcjonalność ma potencjał marketingowy. Kiedy najważniejszym kryterium są pieniądze, poświęcanie czasu na opracowywanie przemyślanych sztuczek optymalizacyjnych nie ma po prostu sensu.

Elegancję programowania pozostawiono hakerom, hobbistom komputerowym, których celem nie jest zarabianie, lecz wyciśnięcie wszystkiego, co się da, z ich starych maszyn Commodore 64, autorom programów wykorzystujących błędy, piszącym niewielkie i niesamowite fragmenty kodu, dzięki którym potrafią przedostawać się przez wąskie szczeliny systemów zabezpieczeń, oraz wszystkim innym doceniającym wartość szukania i znajdowania najlepszych możliwych rozwiązań. Są to osoby zachwycające się możliwościami oferowanymi przez programowanie i dostrzegające piękno eleganckich fragmentów kodu oraz geniusz przemyślanych sztuczek. Poznanie istoty programowania jest niezbędne do zrozumienia, w jaki sposób można wykorzystywać słabości programów, dlatego też programowanie to naturalny punkt wyjścia dla naszych rozważań.

0x210 Istota programowania

Programowanie to pojęcie niezmiernie naturalne i intuicyjne. *Program* jest tylko serią instrukcji zapisanych w określonym języku. Programy występują wszędzie i nawet technofoby codziennie korzystają z programów. Opisywanie drogi dojazdu w jakieś miejsce, przepisy kulinarne, rozgrywki piłkarskie i spirale DNA to są różnego typu programy. Typowy „program" opisu dojazdu samochodem w konkretne miejsce może wyglądać tak:

```
Najpierw zjedź w dół Aleją Kościuszki, prowadzącą na wschód. Jedź nią
do momentu, kiedy zobaczysz po prawej stronie kościół. Jeżeli ulica będzie
zablokowana ze względu na prowadzone roboty, skręć w prawo w ul. Moniuszki,
potem skręć w lewo w ul. Prusa i w końcu skręć w prawo w ul. Orzeszkowej.
W przeciwnym razie możesz po prostu kontynuować jazdę Aleją Kościuszki
i skręcić w ul. Orzeszkowej. Jedź nią i skręć w ul. Docelową. Jedź nią
jakieś 8 kilometrów – nasz dom będzie się znajdował po prawej stronie.
Dokładny adres to ul. Docelowa 743.
```

Każdy, kto zna język polski, pojmie podane instrukcje i będzie mógł kierować się nimi podczas jazdy. Bez wątpienia nie zapisano ich zbyt błyskotliwie, ale są jednoznaczne i zrozumiałe.

Jednak komputer nie zna języka polskiego — rozumie jedynie *język maszynowy* (ang. *machine language*). W celu poinstruowania go, aby wykonał pewną czynność, instrukcję tę należy zapisać w zrozumiałym dla niego języku. Jednakże język maszynowy jest bardzo ezoteryczny i trudny do opanowania. Składa się z serii bitów oraz bajtów i różni się, w zależności od danej architektury komputerowej. Tak więc w celu napisania programu w języku maszynowym dla procesora z rodziny Intel x86 należy określić wartość związaną z każdą instrukcją, sposób interakcji poszczególnych instrukcji oraz mnóstwo innych niskopoziomowych szczegółów. Tego rodzaju programowanie jest bardzo niewdzięcznym oraz kłopotliwym zadaniem i z pewnością nie jest intuicyjne.

Do pokonania komplikacji związanych z pisaniem programów w języku maszynowym potrzebny jest translator. Jedną z form translatora języka maszynowego jest *asembler*. To program, który tłumaczy język asemblera na kod zapisany w języku maszynowym. *Język asemblera* jest bardziej przystępny od kodu maszynowego, ponieważ dla różnych instrukcji i zmiennych wykorzystuje nazwy (*mnemoniki*), a nie same wartości liczbowe. Jednak język asemblera wciąż jest daleki od intuicyjnego. Nazwy instrukcji są ezoteryczne, a sam język wciąż jest zależny od architektury. Oznacza to, że tak samo, jak język maszynowy dla procesorów Intel x86 różni się od języka maszynowego dla procesorów Sparc, język asemblera x86 jest różny od języka asemblera Sparc. Żaden program napisany przy użyciu języka asemblera dla architektury jednego procesora nie będzie działał w architekturze innego procesora. Jeżeli program został napisany w języku asemblera x86, musi zostać przepisany, aby można było uruchomić go w architekturze Sparc. Ponadto w celu napisania wydajnego programu w języku asemblera należy znać wiele niskopoziomowych szczegółów, dotyczących architektury danego procesora.

Problemów tych można uniknąć przy użyciu kolejnej formy translatora, noszącego nazwę *kompilatora* (ang. *compiler*). Kompilator konwertuje kod zapisany w języku wysokopoziomowym do postaci kodu maszynowego. Języki wysokopoziomowe są o wiele bardziej intuicyjne od języka asemblera i mogą być konwertowane do wielu różnych typów języka maszynowego dla odmiennych architektur procesorów. Oznacza to, że jeśli zapisze się program w języku wysokiego poziomu, ten sam kod może zostać skompilowany przez kompilator do postaci kodu maszynowego dla różnych architektur. C, C++ lub FORTRAN to przykłady języków wysokopoziomowych. Program napisany w takim języku jest o wiele bardziej czytelny[1] od języka asemblera lub maszynowego, choć wciąż musi być zgodny z bardzo surowymi regułami dotyczącymi sposobu wyrażania instrukcji, ponieważ w przeciwnym razie kompilator nie będzie w stanie ich zrozumieć.

[1] Oczywiście pod warunkiem, że przez czytelny rozumiemy taki, który przypomina język angielski — *przyp. tłum.*

0x220 Pseudokod

Programiści dysponują jeszcze jednym rodzajem języka programowania, noszącym nazwę pseudokodu. *Pseudokod* (ang. *pseudo-code*) to po prostu wyrażenia zapisane w języku naturalnym, ustawione w strukturę ogólnie przypominającą język wysokopoziomowy. Nie jest on rozumiany przez kompilatory, asemblery ani komputery, ale stanowi przydatny sposób składania instrukcji. Pseudokod nie jest dobrze zdefiniowany. W rzeczywistości wiele osób zapisuje pseudokod odmiennie. Jest to rodzaj nieokreślonego, brakującego ogniwa między językami naturalnymi, takimi jak angielski lub polski, a wysokopoziomowymi językami programowania, takimi jak C lub Pascal. Pseudokod stanowi świetne wprowadzenie do uniwersalnych założeń programistycznych.

0x230 Struktury sterujące

Bez struktur sterujących program byłby jedynie serią kolejno wykonywanych instrukcji. Jest to wystarczające w przypadku bardzo prostych programów, ale większość programów, takich jak np. opis dojazdu, taka nie jest. Prezentowany opis dojazdu zawierał instrukcje: *Jedź nią do momentu, kiedy zobaczysz po prawej stronie kościół. Jeżeli ulica będzie zablokowana ze względu na prowadzone roboty…* Instrukcje te nazywamy *strukturami sterującymi*, ponieważ zmieniają przebieg wykonywania programu z prostego, sekwencyjnego, na bardziej złożony i użyteczny.

0x231 If-Then-Else

W naszym opisie dojazdu Aleja Kościuszki może być w remoncie. Do rozwiązania problemu wynikającego z tej sytuacji potrzebujemy specjalnego zestawu instrukcji. Jeżeli zdefiniowane okoliczności nie wystąpią (remont), będą wykonywane standardowe instrukcje. Tego rodzaju specjalne przypadki można wziąć pod uwagę w programie, korzystając z jednej z najbardziej naturalnych struktur sterujących: `if-then-else` (jeżeli-wówczas-w przeciwnym przypadku). Ogólnie wygląda to mniej więcej tak:

```
If (warunek) then
{
Zestaw instrukcji do wykonania, gdy warunek jest spełniony;
}
Else
{
Zestaw instrukcji do wykonania, gdy warunek nie jest spełniony;
}
```

W tej książce będziemy posługiwali się pseudokodem przypominającym język C, więc każda instrukcja będzie zakończona średnikiem, a ich zbiory będą grupowane za pomocą nawiasów klamrowych i wyróżniane wcięciem. Pseudokod struktury `if-then-else` dla wskazówek dojazdu mógłby przedstawiać się następująco:

```
Jedź w dół Aleją Kościuszki;
Jeżeli (ulica zablokowana)
{
  Skręć w prawo w ul. Moniuszki;
  Skręć w lewo w ul. Prusa;
  Skręć w prawo w ul. Orzeszkowej;
}
else
{
  Skręć w prawo w ul. Orzeszkowej;
}
```

Każda instrukcja znajduje się w osobnym wierszu, a różne zestawy instrukcji warunkowych są zgrupowane wewnątrz nawiasów klamrowych i dla zwiększenia czytelności zastosowano dla nich wcięcie. W C i wielu innych językach programowania słowo kluczowe `then` jest domniemane i tym samym pomijane, więc pominęliśmy je również w pseudokodzie.

Oczywiście, istnieją języki, których składnia wymaga zastosowania słowa kluczowego `then` — przykładami mogą tu być BASIC, Fortran, a nawet Pascal. Tego typu różnice syntaktyczne między językami programowania są niemal nieistotne, jako że podstawowa struktura jest taka sama. Gdy programista zrozumie założenia tych języków, nauczenie się różnych odmian syntaktycznych jest stosunkowo łatwe. Ponieważ w dalszych rozdziałach będziemy programowali w języku C, prezentowany pseudokod będzie przypominał ten język. Należy jednak pamiętać, że może on przyjmować różne postaci.

Inną powszechną regułą składni podobnej do C jest sytuacja, w której zestaw instrukcji umieszczonych w nawiasach klamrowych zawiera tylko jedną instrukcję. Dla zwiększenia czytelności wciąż warto dla takiej instrukcji zastosować wcięcie, ale nie jest ono wymagane. Zgodnie z tą regułą można więc napisać wcześniej przedstawiony opis dojazdu tak, aby uzyskać równoważny fragment pseudokodu:

```
Jedź w dół Aleją Kościuszki;
Jeżeli (ulica zablokowana)
{
  Skręć w prawo w ul. Moniuszki;
  Skręć w lewo w ul. Prusa;
  Skręć w prawo w ul. Orzeszkowej;
}
else
  Skręć w prawo w ul. Orzeszkowej;
```

Reguła mówiąca o zestawach instrukcji dotyczy wszystkich struktur sterujących opisywanych w niniejszej książce, a samą regułę również można zapisać w pseudokodzie.

```
If (w zestawie instrukcji znajduje się tylko jedna instrukcja)
  Użycie nawiasów klamrowych do zgrupowania instrukcji jest opcjonalne;
Else
{
  Użycie nawiasów klamrowych jest niezbędne;
  Ponieważ musi istnieć logiczny sposób zgrupowania tych instrukcji;
}
```

Nawet opis składni można potraktować jak program. Istnieje wiele odmian struktury `if-then-else`, np. instrukcje `select` i `case`, ale logika pozostaje taka sama: jeżeli stanie się tak, zrób to i to, w przeciwnym przypadku wykonaj, co poniżej (a poniżej mogą się znajdować kolejne instrukcje `if-then`).

0x232 Pętle While/Until

Innym podstawowym pojęciem programistycznym jest struktura sterująca `while`, będąca rodzajem pętli. Programista często chce wykonać zestaw instrukcji kilka razy. Program może wykonać to zadanie, stosując pętlę, ale wymaga to określenia warunków jej zakończenia, aby nie trwała w nieskończoność. Pętla `while` wykonuje podany zestaw instrukcji, dopóki warunek jest prawdziwy. Prosty program dla głodnej myszy mógłby wyglądać następująco:

```
While (jesteś głodna)
{
  Znajdź jakieś jedzenie;
  Zjedz jedzenie;
}
```

Te dwa zestawy instrukcji za instrukcją `while` będą powtarzane, dopóki mysz będzie głodna. Ilość pożywienia odnalezionego przez mysz przy każdej próbie może wahać się od małego okruszka po cały bochen chleba. Podobnie liczba wykonań zestawu instrukcji w pętli `while` zależy od ilości pożywienia znalezionego przez mysz.

Odmianą pętli `while` jest pętla `until`, wykorzystywana w Perlu (w C składnia ta nie jest używana). Pętla `until` jest po prostu pętlą `while` z odwróconym warunkiem. Ten sam program dla myszy zapisany z użyciem pętli `until` wyglądałby następująco:

```
Until (nie jesteś głodna)
{
  Znajdź jakieś jedzenie;
  Zjedz jedzenie;
}
```

Logicznie każda instrukcja podobna do `until` może być przekształcona w pętlę `while`. Opis dojazdu zawierał zdanie: „Jedź nią do momentu, kiedy zobaczysz po prawej stronie kościół". Możemy to w prosty sposób przekształcić w standardową pętlę `while` — wystarczy, że odwrócimy warunek:

```
While (nie ma kościoła po prawej stronie)
  Jedź Aleją Kościuszki;
```

0x233 Pętle for

Kolejną strukturą sterującą jest pętla `for`, wykorzystywana wtedy, gdy programista chce, aby pętla wykonała określoną liczbę iteracji. Instrukcja dojazdu: „Jedź nią jakieś 8 kilometrów" przekształcona do pętli `for` mogłaby wyglądać następująco:

```
For (8 iteracji)

  Jedź prosto 1 kilometr;
```

W rzeczywistości pętla `for` jest po prostu pętlą `while` z licznikiem. Powyższą instrukcję można równie dobrze zapisać tak:

```
Ustaw licznik na 0
While (licznik jest mniejszy od 8)
{
  Jedź prosto 1 kilometr;
  Dodaj 1 do licznika;
}
```

Składnia pseudokodu dla pętli `for` sprawia, że jest to jeszcze bardziej widoczne:

```
For (i=0; i<8; i++)
  Jedź prosto 1 kilometr;
```

W tym przypadku licznik nosi nazwę `i`, a instrukcja `for` jest podzielona na trzy sekcje oddzielone średnikami. Pierwsza sekcja deklaruje licznik i ustawia jego początkową wartość — w tym przypadku 0. Druga sekcja przypomina instrukcję `while` z licznikiem — dopóki licznik spełnia podany warunek, kontynuuj pętlę. Trzecia i ostatnia sekcja opisuje, co ma się dziać z licznikiem przy każdej iteracji. W tym przypadku `i++` jest skróconym sposobem na powiedzenie: „Dodaj 1 do licznika o nazwie i".

Zastosowanie wszystkich omówionych struktur sterujących umożliwia przekształcenie opisu dojazdu ze strony 6 w poniższy pseudokod przypominający język C:

```
Rozpocznij jazdę na wschód Aleją Kościuszki;
While (po prawej stronie nie ma kościoła)
   Jedź Aleją Kościuszki;
If (ulica zablokowana)
{
   Skręć w prawo w ul. Moniuszki);
   Skręć w lewo w ul. Prusa);
   Skręć w prawo w ul. Orzeszkowej);
}
```

```
Else
   Skręć(prawo, ul. Orzeszkowej);
Skręć(lewo, ul. Docelowa);
For (i=0; i<8; i++)
{
   Jedź prosto 1 kilometr;
}
Zatrzymaj się przy ul. Docelowej 743;
```

0x240 Podstawowe pojęcia programistyczne

W kolejnych punktach zostaną przedstawione podstawowe pojęcia programistyczne. Są one wykorzystywane w wielu językach programowania, różnią się jedynie pod względem syntaktycznym. Podczas prezentacji będą integrowane z przykładami w pseudokodzie przypominającym składnię języka C. Na końcu pseudokod będzie bardzo przypominał kod napisany w języku C.

0x241 Zmienne

Licznik wykorzystywany w pętli `for` jest rodzajem zmiennej. O *zmiennej* możemy myśleć jak o obiekcie przechowującym dane, które mogą być zmieniane — stąd nazwa. Istnieją również zmienne, których wartości nie są zmieniane, i trafnie nazywa się je *stałymi*. W przykładzie motoryzacyjnym jako zmienną możemy potraktować prędkość samochodu, natomiast kolor lakieru samochodu byłby stałą. W pseudokodzie zmienne są po prostu pojęciami abstrakcyjnymi, natomiast w C (i wielu innych językach) zmienne przed wykorzystaniem muszą być zadeklarowane i należy określić ich typ. Jest tak dlatego, że program napisany w C będzie kompilowany do postaci programu wykonywalnego. Podobnie jak przepis kucharski, w którym przed faktycznymi instrukcjami wymieniane są wszystkie potrzebne składniki, deklaracje zmiennych pozwalają dokonać przygotowań przed przejściem do faktycznej treści programu. Ostatecznie wszystkie zmienne są przechowywane gdzieś w pamięci, a ich deklaracje pozwalają kompilatorowi wydajniej uporządkować pamięć. Jednak na koniec, mimo wszystkich deklaracji typów zmiennych, wszystko odbywa się w pamięci.

W języku C dla każdej zmiennej określany jest typ, opisujący informacje, które będą przechowywane w tej zmiennej. Do najczęściej stosowanych typów zaliczamy `int` (liczby całkowite), `float` (liczby zmiennoprzecinkowe) i `char` (pojedyncze znaki). Zmienne są deklarowane poprzez użycie tych słów kluczowych przed nazwą zmiennej, co zobrazowano na poniższym przykładzie.

```
int a, b;
float k;
char z;
```

Zmienne `a` i `b` są zadeklarowane jako liczby całkowite, `k` może przyjmować wartości zmiennoprzecinkowe (np. 3,14), a `z` powinna przechowywać wartość znakową,

taką jak A lub w. Wartości można przypisywać zmiennym podczas deklaracji lub w dowolnym późniejszym momencie. Służy do tego operator =.

```
int a = 13, b;
float k;
char z = 'A';

k = 3.14;
z = 'w';
b = a + 5;
```

Po wykonaniu powyższych instrukcji zmienna a będzie posiadała wartość 13, zmienna k będzie zawierała liczbę 3,14, a zmienna b będzie miała wartość 18, ponieważ 13 plus 5 równa się 18. Zmienne są po prostu sposobem pamiętania wartości, jednak w języku C należy najpierw zadeklarować typ każdej zmiennej.

0x242 Operatory arytmetyczne

Instrukcja `b = a + 7` jest przykładem bardzo prostego operatora arytmetycznego. W języku C dla różnych operacji matematycznych wykorzystywane są poniższe symbole.

Pierwsze cztery operatory powinny wyglądać znajomo. Redukcja modulo może stanowić nowe pojęcie, ale jest to po prostu reszta z dzielenia. Jeżeli a ma wartość 13, to 13 podzielone przez 5 równa się 2, a reszta wynosi 3, co oznacza, że `a % 5 = 3`. Ponadto skoro zmienne a i b są liczbami całkowitymi, po wykonaniu instrukcji `b = a / 5`, zmienna b będzie miała wartość 2, ponieważ jest to część wyniku będąca liczbą całkowitą. Do przechowania bardziej poprawnego wyniku, czyli 2,6, musi zostać użyta zmienna typu float (zmiennoprzecinkowa).

Operacja	Symbol	Przykład
Dodawanie	+	`b = a + 5`
Odejmowanie	-	`b = a - 5`
Mnożenie	*	`b = a * 5`
Dzielenie	/	`b = a / 5`
Redukcja modulo	%	`b = a % 5`

Aby program korzystał z tych założeń, musimy mówić w jego języku. Język C dostarcza kilku skróconych form zapisu tych operacji arytmetycznych. Jedna z nich została przedstawiona wcześniej i jest często wykorzystywana w pętlach for.

Pełne wyrażenie	Skrót	Wyjaśnienie
`i = i + 1`	`i++` lub `++i`	Dodaje 1 do wartości zmiennej.
`i = i - 1`	`i--` lub `i--`	Odejmuje 1 od wartości zmiennej.

Te skrócone wyrażenia mogą być łączone z innymi operacjami arytmetycznymi w celu utworzenia bardziej złożonych wyrażeń. Wówczas staje się widoczna różnica między `i++` i `++i`. Pierwsze wyrażenie oznacza: „Zwiększ wartość `i` o 1 po przeprowadzeniu operacji arytmetycznej", natomiast drugie: „Zwiększ wartość `i` o 1 przed przeprowadzeniem operacji arytmetycznej". Poniższy przykład pozwoli to lepiej zrozumieć.

```
int a, b;
a = 5
b = a++ * 6;
```

Po wykonaniu tego zestawu instrukcji zmienna `b` będzie miała wartość 30, natomiast zmienna `a` będzie miała wartość 6, ponieważ skrócony zapis `b = a++ * 6` jest równoważny poniższym instrukcjom.

```
b = a * 6;
a = a + 1;
```

Jeżeli jednak zostanie użyta instrukcja `b = ++a * 6`, kolejność dodawania zostanie zmieniona i otrzymamy odpowiednik poniższych instrukcji.

```
a = a + 1;
b = a * 6;
```

Ponieważ kolejność wykonywania działań zmieniła się, w powyższym przypadku zmienna `b` będzie miała wartość 36, natomiast wartość zmiennej `a` wciąż będzie wynosiła 6.

Często w programach musimy zmodyfikować zmienną w danym miejscu. Przykładowo możemy dodać dowolną wartość, taką jak 12, i przechować wynik w tej samej zmiennej (np. `i = i + 12`) Zdarza się to tak często, że również utworzono skróconą formę tej instrukcji.

Pełne wyrażenie	Skrót	Wyjaśnienie
`i = i + 12`	`i+=12`	Dodaje określoną wartość do wartości zmiennej.
`i = i - 12`	`i-=12`	Odejmuje określoną wartość od wartości zmiennej.
`i = i * 12`	`i*=12`	Mnoży określoną wartość przez wartość zmiennej.
`i = i / 12`	`i/=12`	Dzieli wartość zmiennej przez określoną wartość.

0x243 Operatory porównania

Zmienne są często wykorzystywane w instrukcjach warunkowych wcześniej opisanych struktur sterujących, a te instrukcje warunkowe opierają się na jakiegoś rodzaju porównaniu. W języku C operatory porównania wykorzystują skróconą składnię, która jest stosunkowo powszechna w innych językach programowania.

Warunek	Symbol	Przykład
Mniejsze niż	<	(a < b)
Większe niż	>	(a < b)
Mniejsze lub równe	<=	(a <= b)
Większe lub równe	>=	(a >= b)
Równe	==	(a == b)
Nie równa się	!=	(a != b)

Większość z tych operatorów jest zrozumiała, jednakże należy zauważyć, że w skrócie porównania równości wykorzystywane są dwa znaki równości. Jest to ważne rozróżnienie, ponieważ podwójny znak równości jest stosowany do sprawdzenia równości, natomiast jeden znak równości jest używany do przypisania wartości zmiennej. Instrukcja a = 7 oznacza: „Umieść w zmiennej a wartość 7", natomiast a == 7 oznacza: „Sprawdź, czy wartość zmiennej a wynosi 7". (Niektóre języki programowania, takie jak Pascal, wykorzystują do przypisywania wartości zmiennym następujący znak — := — aby wyeliminować ryzyko błędnego zrozumienia kodu). Należy też zwrócić uwagę, że wykrzyknik zwykle oznacza nie. Ten symbol może być wykorzystany do odwrócenia dowolnego wyrażenia.

```
!(a < b)   jest równoważne   (a >= b)
```

Operatory porównania mogą być łączone ze sobą za pomocą skrótów dla operatorów logicznych OR i AND.

Operator logiczny	Symbol	Przykład
OR	||	((a < b) || (a < c))
AND	&&	((a < b) && !(a < c))

Przykładowa instrukcja zawierająca dwa mniejsze warunki połączone operatorem OR zostanie oszacowana jako prawdziwa, jeżeli wartość zmiennej a będzie mniejsza od wartości zmiennej b LUB wartość zmiennej a będzie mniejsza od wartości zmiennej c. Podobnie przykładowa instrukcja zawierająca dwa mniejsze warunki połączone operatorem AND zostanie oszacowana jako prawdziwa, jeżeli wartość zmiennej a będzie mniejsza od wartości zmiennej b I wartość zmiennej a nie będzie mniejsza od wartości zmiennej c. Takie instrukcje powinny być grupowane za pomocą nawiasów i mogą zawierać wiele różnych odmian.

Wiele rzeczy możemy sprowadzić do zmiennych, operatorów porównania i struktur sterujących. W przykładzie myszy poszukującej pożywienia głód możemy przedstawić jako zmienną boolowską o wartości true lub false (prawda lub fałsz). Oczywiście, 1 oznacza true, a 0 — false.

```
While (głód == 1)
{
  Szukaj pożywienia;
  Zjedz pożywienie;
}
```

Oto kolejny skrót często wykorzystywany przez programistów i hakerów. Język C nie zawiera żadnych operatorów boolowskich, więc każda wartość niezerowa jest uznawana za `true`, a instrukcja jest szacowana jako `false`, gdy zawiera 0. W rzeczywistości operatory porównania faktycznie zwracają 1, jeżeli porównanie jest spełnione, a w przeciwnym przypadku zwracają 0. Sprawdzenie, czy zmienna `głód` ma wartość 1, zwróci 1, jeżeli `głód` ma wartość 1, lub zwróci 0, jeżeli głód ma wartość 0. Ponieważ program wykorzystuje tylko te dwa przypadki, można w ogóle pominąć operator porównania.

```
While (głód)
{
  Szukaj pożywienia;
  Zjedz pożywienie;
}
```

Sprytniejszy program sterujący zachowaniem myszy z użyciem większej liczby danych wejściowych to przykład, jak można łączyć operatory porównania ze zmiennymi.

```
While ((głód) && !(kot_obecny))
{
  Szukaj pożywienia;
  If (!(pożywienie_znajduje_się_w_pułapce_na_myszy;
    Zjedz pożywienie;
}
```

W tym przykładzie założono, że dostępne są również zmienne opisujące obecność kota oraz położenie pożywienia z wartościami 1 dla spełnienia warunku i 0 dla przypadku przeciwnego. Wystarczy pamiętać, że wartość niezerowa oznacza prawdę, a wartość 0 jest szacowana jako fałsz.

0x244 Funkcje

Czasami zdarzają się zestawy instrukcji, których programista będzie potrzebował kilka razy. Takie instrukcje można zgrupować w mniejszym podprogramie, zwanym *funkcją*. W innych językach funkcje nazywane są także procedurami i podprocedurami. Przykładowo wykonanie skrętu samochodem składa się z kilku mniejszych instrukcji: włącz odpowiedni kierunkowskaz, zwolnij, sprawdź, czy nie nadjeżdżają inne samochody, skręć kierownicę w odpowiednim kierunku itd. Opis trasy z początku tego rozdziału zawierał całkiem sporo skrętów, jednakże wymienianie wszystkich

drobnych instrukcji przy każdym skręcie byłoby żmudne (i zmniejszałoby czytelność kodu). W celu zmodyfikowania sposobu działania funkcji możemy przesyłać do niej zmienne jako argumenty. W tym przypadku do funkcji jest przesyłany kierunek skrętu.

```
Function Skręt(zmienna_kierunku)
{
  Włącz zmienna_kierunku kierunkowskaz;
  Zwolnij;
  Sprawdź, czy nie nadjeżdżają inne samochody;
  while (nadjeżdżają inne samochody)
  {
    Zatrzymaj się;
    Obserwuj nadjeżdżające samochody;
  }
  Obróć kierownicę w zmienna_kierunku;
  while (skręt nie jest ukończony)
  {
    if (prędkość < 8 km/h)
      Przyspiesz;
  }
  Obróć kierownicę do pierwotnego położenia;
  Wyłącz zmienna_kierunku kierunkowskaz;
}
```

Funkcja ta zawiera wszystkie instrukcje konieczne do wykonania skrętu. Gdy program, w którym funkcja taka się znajduje, musi wykonać skręt, może po prostu wywołać tę funkcję. Gdy zostaje wywołana, znajdujące się w niej instrukcje są wykonywane z przesłanymi argumentami. Następnie wykonywanie programu powraca do momentu za wywołaniem funkcji. Do tej funkcji można przesłać wartość lewo albo prawo, co spowoduje wykonanie skrętu w tym kierunku.

Domyślnie w języku C funkcje mogą zwracać wartości do elementu wywołującego. Dla osób znających matematykę jest to zupełnie zrozumiałe. Wyobraźmy sobie funkcję obliczającą silnię — jest oczywiste, że zwraca wynik.

W języku C funkcje nie są oznaczane słowem kluczowym `function`. Zamiast tego są deklarowane przez typ danych zwracanej zmiennej. Taki format bardzo przypomina deklarację zmiennej. Jeżeli funkcja w zamierzeniu ma zwracać liczbę całkowitą (może to być np. funkcja obliczająca silnię jakiejś liczby `x`), jej kod mógłby wyglądać następująco:

```
int factorial(int x)
{
  int i;
  for(i=1; i < x; i++)
    x *= i;
  return x;
}
```

Funkcja ta jest deklarowana liczbą całkowitą, ponieważ mnoży każdą wartość od 1 do `x` i zwraca wynik, który jest liczbą całkowitą. Instrukcja `return` na końcu funkcji przesyła zawartość zmiennej `x` i kończy wykonywanie funkcji. Funkcję obliczającą silnię można wówczas wykorzystać jak każdą zmienną typu `integer` w głównej części każdego programu, który „wie" o tej funkcji:

```
int a=5, b;
b = factorial(a);
```

Po ukończeniu wykonywania tego krótkiego programu zmienna `b` będzie miała wartość 120, ponieważ funkcja `factorial()` zostanie wywołana z argumentem 5 i zwróci 120.

Ponadto w języku C kompilator musi „wiedzieć" o funkcji, zanim jej użyje. W tym celu można po prostu napisać całą funkcję przed późniejszym wykorzystaniem jej w programie lub skorzystać z prototypu funkcji. *Prototyp funkcji* jest prostym sposobem poinformowania kompilatora, iż powinien spodziewać się funkcji o podanej nazwie, zwracającej określony typ danych i przyjmującej argumenty o określonym typie danych. Faktyczny kod funkcji można umieścić pod koniec programu, ale można ją wykorzystać w dowolnym miejscu, ponieważ kompilator będzie już wiedział o jej istnieniu. Przykładowy prototyp funkcji `factorial()` mógłby wyglądać następująco:

```
int factorial(int);
```

Zwykle prototypy funkcji są umieszczane blisko początku programu. W prototypie nie ma potrzeby definiowania nazw zmiennych, ponieważ dzieje się to w samej funkcji. Kompilator musi jedynie znać nazwę funkcji, typ zwracanych danych i typy danych argumentów funkcjonalnych.

Jeżeli funkcja nie posiada wartości, którą może zwrócić (np. wcześniej opisana funkcja `skręć()`), powinna być zadeklarowana jako typ `void`. Jednakże funkcja `skręć()` nie posiada jeszcze całej funkcjonalności wymaganej przez nasz opis dojazdu. Każdy skręt w opisie dojazdu zawiera zarówno kierunek, jak i nazwę ulicy. To oznacza, że funkcja skręcająca powinna przyjmować dwie zmienne: kierunek skrętu oraz nazwę ulicy, w którą należy skręcić. To komplikuje funkcję skręcającą, ponieważ przed wykonaniem skrętu należy zlokalizować ulicę. Bardziej kompletna funkcja skręcająca, w której wykorzystano składnię podobną do języka C, jest zamieszczona w pseudokodzie poniższego listingu.

```
void skręć(zmienna_kierunku, nazwa_docelowej_ulicy)
{
  Szukaj tabliczki z nazwą ulicy;
  nazwa_bieżącego_skrzyżowania = odczytaj tabliczkę z nazwą ulicy;
  while (nazwa_bieżącego_skrzyżowania != nazwa_docelowej_ulicy)
  {
    Szukaj innej tabliczki z nazwą ulicy;
    nazwa_bieżącego_skrzyżowania = odczytaj tabliczkę z nazwą ulicy;
  }
```

```
Włącz zmienna_kierunku kierunkowskaz;
Zwolnij;
Sprawdź, czy nie nadjeżdżają inne samochody;
  while (nadjeżdżają inne samochody)
  {
    Zatrzymaj się;
    Obserwuj nadjeżdżające samochody;
  }
  Obróć kierownicę w zmienna_kierunku;
  while (skręt nie jest ukończony)
  {
    if (prędkość < 8 km/h)
      Przyspiesz;
  }
  Obróć kierownicę do pierwotnego położenia;
  Wyłącz zmienna_kierunku kierunkowskaz;
}
```

Funkcja ta zawiera sekcję, która wyszukuje odpowiednie skrzyżowanie poprzez odnajdywanie tabliczki z nazwą ulicy, odczytywanie z niej nazwy i przechowywanie jej w zmiennej o nazwie `nazwa_bieżącego_skrzyżowania`. Wyszukiwanie i odczytywanie tabliczek z nazwami odbywa się do momentu odnalezienia docelowej ulicy. Wówczas wykonywane są pozostałe instrukcje skrętu. Możemy teraz zmienić pseudokod z instrukcjami dojazdu, aby wykorzystywał funkcję skrętu.

```
Rozpocznij jazdę na wschód Aleją Kościuszki;
while (po prawej stronie nie ma kościoła)
{
   Jedź Aleją Kościuszki;
}
If (ulica zablokowana)
{
   Skręć(prawo, ul. Moniuszki);
   Skręć(lewo, ul. Prusa);
   Skręć(prawo, ul. Orzeszkowej);
}
else
{
   Skręć(prawo, ul. Orzeszkowej);
}
Skręć(lewo, ul. Docelowa);
For (i=0; i<5; i++)
{
   Jedź prosto 1 kilometr;
}
Zatrzymaj się przy ul. Docelowej 743;
```

Funkcje nie są często wykorzystywane w pseudokodzie, bo zwykle jest on stosowany przez programistów do zarysowania założeń programu przed napisaniem kompilowalnego kodu. Ponieważ pseudokod nie musi działać, nie trzeba wypisywać pełnych funkcji — wystarczy jedynie krótki zapisek: *Tu zrób coś złożonego*. Jednak

w języku programowania, takim jak C, funkcje są często wykorzystywane. Większość prawdziwej użyteczności C pochodzi ze zbiorów istniejących funkcji, zwanych *bibliotekami*.

0x250 Zaczynamy brudzić sobie ręce

Gdy zaznajomiliśmy się już trochę ze składnią języka C i wyjaśniliśmy kilka podstawowych pojęć programistycznych, przejście do faktycznego programowania w C nie jest niczym nadzwyczajnym. Kompilatory języka C istnieją dla niemal każdego systemu operacyjnego i architektury procesora, jednak w niniejszej książce będziemy wykorzystywali wyłącznie system Linux i procesor z rodziny x86. Linux jest darmowym systemem operacyjnym, do którego każdy ma dostęp, a procesory x86 są najpopularniejszymi procesorami konsumenckimi na świecie. Ponieważ hacking wiąże się głównie z eksperymentowaniem, najlepiej będzie, jeżeli w trakcie lektury będziesz dysponował kompilatorem języka C.

Dzięki dyskowi LiveCD dołączonemu do niniejszej książki możesz praktycznie stosować przekazywane informacje, jeżeli tylko dysponujesz procesorem z rodziny x86. Wystarczy włożyć dysk CD do napędu i ponownie uruchomić komputer. Uruchomione zostanie środowisko linuksowe, które nie narusza istniejącego systemu operacyjnego. W tym środowisku możesz wypróbowywać przykłady z książki, a także przeprowadzać własne eksperymenty.

Przejdźmy do rzeczy. Program *firstprog.c* jest prostym fragmentem kodu w C, wypisującym 10 razy „Hello world!".

Listing 2.1. *firstprog.c*

```
#include <stdio.h>

int main()
{
  int i;
  for(i=0; i < 10; i++) // Wykonaj pętlę 10 razy.
  {
    puts("Hello, world!\n"); // Przekaż łańcuch do wyjścia.
  }
  return 0; // Poinformuj OS, że program zakończył się bez błędów.
}
```

Główne wykonywanie programu napisanego w C rozpoczyna się w funkcji `main()`. Tekst znajdujący się za dwoma ukośnikami jest komentarzem ignorowanym przez kompilator.

Pierwszy wiersz może wprawiać w zakłopotanie, ale jest to jedynie składnia informująca kompilator, aby dołączył nagłówki dla standardowej biblioteki wejścia-wyjścia o nazwie *stdio*. Ten plik nagłówkowy jest dodawany do programu podczas kompilacji. Znajduje się on w */usr/include/stdio.h* i definiuje kilka stałych i prototypów funkcji dla odpowiednich funkcji w standardowej bibliotece we-wy. Ponieważ funkcja `main()` wykorzystuje funkcję `printf()` ze standardowej biblioteki

we-wy, wymagany jest prototyp funkcji `printf()` przed jej zastosowaniem. Ten prototyp funkcji (a także wiele innych) znajduje się w pliku nagłówkowym *stdio.h*. Sporo możliwości języka C to pochodne jego zdolności do rozbudowy oraz wykorzystania bibliotek. Reszta kodu powinna być zrozumiała i przypomina wcześniej prezentowany pseudokod. Są nawet nawiasy klamrowe, które można było pominąć. To, co będzie się działo po uruchomieniu tego programu, powinno być oczywiste, ale skompilujmy go za pomocą GCC, aby się upewnić.

GNU Compiler Collection jest darmowym kompilatorem języka C, tłumaczącym język C na język maszynowy, zrozumiały dla komputera. W wyniku tłumaczenia powstaje wykonywalny plik binarny, któremu domyślnie nadawana jest nazwa *a.out*. Czy skompilowany program wykonuje to, co chcieliśmy?

```
reader@hacking:~/booksrc $ gcc firstprog.c
reader@hacking:~/booksrc $ ls -l a.out
-rwxr-xr-x 1 reader reader 6621 2007-09-06 22:16 a.out
reader@hacking:~/booksrc $ ./a.out
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
reader@hacking:~/booksrc $
```

0x251 Większy obraz

Dobrze, tego wszystkiego można się nauczyć na podstawowym kursie programowania — są to podstawy, ale niezbędne. Większość początkowych kursów programowania uczy jedynie, jak czytać i pisać w języku C. Proszę mnie źle nie zrozumieć — płynna znajomość C jest bardzo przydatna i wystarczy do stania się dobrym programistą, ale jest to tylko część większej całości. Większość programistów uczy się języka z góry do dołu i nigdy nie ma szerszego obrazu. Hakerzy są tak dobrzy, ponieważ wiedzą, jak wszystkie części tego większego obrazu wchodzą ze sobą w interakcje. Aby w programowaniu patrzeć szerzej, wystarczy sobie zdać sprawę, że kod w języku C będzie kompilowany. Kod nie może nic zrobić, dopóki nie zostanie skompilowany do wykonywalnego pliku binarnego. Myślenie o kodzie źródłowym w C jak o programie jest częstym błędem wykorzystywanym codziennie przez hakerów. Instrukcje w pliku *a.out* są zapisane w języku maszynowym — podstawowym języku zrozumiałym dla procesora. Kompilatory tłumaczą język kodu C na język maszynowy różnych architektur procesorów. W tym przypadku procesor pochodzi z rodziny wykorzystującej architekturę x86. Istnieją również architektury procesorów Sparc (używane w stacjach roboczych Sun) oraz architektura PowerPC

(wykorzystywana w macintoshach przed zastosowaniem procesorów Intela). Każda architektura wymaga innego języka maszynowego, więc kompilator jest pośrednikiem — tłumaczy kod w języku C na język maszynowy docelowej architektury.

Dopóki skompilowany program działa, przeciętny programista skupia się wyłącznie na kodzie źródłowym. Jednak haker zdaje sobie sprawę, że w rzeczywistości wykonywany jest program skompilowany. Mając dogłębną wiedzę na temat działania procesora, haker może manipulować działającymi programami. Widzieliśmy kod źródłowy naszego pierwszego programu i skompilowaliśmy go do pliku wykonywalnego dla architektury x86. Ale jak wygląda ten wykonywalny plik binarny? Narzędzia programistyczne GNU zawierają program o nazwie *objdump*, który może być użyty do badania skompilowanych plików binarnych. Rozpocznijmy od poznania kodu maszynowego, na który została przetłumaczona funkcja `main()`.

```
reader@hacking:~/booksrc $ objdump -D a.out | grep -A20 main.:
08048374 <main>:
8048374: 55 push %ebp
8048375: 89 e5 mov %esp,%ebp
8048377: 83 ec 08 sub $0x8,%esp
804837a: 83 e4 f0 and $0xfffffff0,%esp
804837d: b8 00 00 00 00 mov $0x0,%eax
8048382: 29 c4 sub %eax,%esp
8048384: c7 45 fc 00 00 00 00 movl $0x0,0xfffffffc(%ebp)
804838b: 83 7d fc 09 cmpl $0x9,0xfffffffc(%ebp)
804838f: 7e 02 jle 8048393 <main+0x1f>
8048391: eb 13 jmp 80483a6 <main+0x32>
8048393: c7 04 24 84 84 04 08 movl $0x8048484,(%esp)
804839a: e8 01 ff ff ff call 80482a0 <printf@plt>
804839f: 8d 45 fc lea 0xfffffffc(%ebp),%eax
80483a2: ff 00 incl (%eax)
80483a4: eb e5 jmp 804838b <main+0x17>
80483a6: c9 leave
80483a7: c3 ret
80483a8: 90 nop
80483a9: 90 nop
80483aa: 90 nop
reader@hacking:~/booksrc $
```

Program *objdump* zwrócił zbyt wiele wierszy wynikowych, aby rozsądnie się im przyjrzeć, więc przekazaliśmy wyjście do `grep` z opcją powodującą wyświetlenie tylko 20 wierszy za wyrażeniem regularnym `main.:`. Każdy bajt jest reprezentowany w *notacji szesnastkowej*, systemie liczbowym o podstawie 16. System liczbowy, do którego jesteśmy najbardziej przyzwyczajeni, wykorzystuje podstawę 10, a przy liczbie 10 należy umieścić dodatkowy symbol. W systemie szesnastkowym stosowane są cyfry od *0* do 9 reprezentujące 0 do 9, ale także litery *A* do *F*, które reprezentują w nim liczby od 10 do 15. Jest to wygodny zapis, ponieważ bajt zawiera 8 bitów, z których każdy może mieć wartość prawda lub fałsz. To oznacza, że bajt ma 256 (2^8) możliwych wartości, więc każdy bajt może być opisany za pomocą dwóch cyfr w systemie szesnastkowym.

Liczby szesnastkowe — rozpoczynając od 0x8048374 po lewej stronie — są adresami pamięci. Bity języka maszynowego muszą być gdzieś przechowywane, a tym „gdzieś" jest *pamięć*. Pamięć jest po prostu zbiorem bajtów z tymczasowej przestrzeni magazynowej, którym przypisano adresy liczbowe.

Podobnie jak rząd domów przy lokalnej ulicy, z których każdy posiada własny adres, pamięć możemy traktować jako rząd bajtów, z których każdy ma własny adres. Do każdego bajta pamięci można uzyskać dostęp za pomocą jego adresu, a w tym przypadku procesor sięga do tej części pamięci, aby pobrać instrukcje języka maszynowego składające się na skompilowany program. Starsze procesory Intela z rodziny x86 wykorzystują 32-bitowy schemat adresacji, natomiast nowsze — 64-bitowy. Procesory 32-bitowe posiadają 2^{32} (lub też 4 294 967 296) możliwych adresów, natomiast procesory 64-bitowe dysponują ilością adresów równą 2^{64} ($1{,}84467441 \times 10^{19}$). Procesory 64-bitowe mogą pracować w trybie kompatybilności 32-bitowej, co umożliwia im szybkie wykonywanie kodu 32-bitowego.

Bajty szesnastkowe znajdujące się pośrodku powyższego listingu są instrukcjami w języku maszynowym procesora x86. Oczywiście, te wartości szesnastkowe są jedynie reprezentacjami binarnych jedynek i zer, które rozumie procesor. Ale ponieważ 01010101100010011110010110000011111011001111000 01... przydaje się tylko procesorowi, kod maszynowy jest wyświetlany jako bajty szesnastkowe, a każda instrukcja jest umieszczana w osobnym wierszu, co przypomina podział akapitu na zdania.

Gdy się zastanowimy, okaże się, że bajty szesnastkowe same w sobie też nie są przydatne — tutaj dochodzi do głosu język asembler. Instrukcje po prawej stronie są zapisane w asemblerze. Asembler jest po prostu zbiorem mnemoników dla odpowiednich instrukcji języka maszynowego. Instrukcja `ret` jest znacznie łatwiejsza do zapamiętania i zrozumienia niż 0xc3 czy też 11000011. W przeciwieństwie do C i innych języków kompilowanych, instrukcje asemblera występują w relacji jeden-do-jednego z instrukcjami odpowiednimi języka maszynowego. Oznacza to, że skoro każda architektura procesora posiada własny zestaw instrukcji języka maszynowego, każda z nich ma również własną odmianę asemblera. Język asemblera jest dla programistów tylko sposobem reprezentacji instrukcji języka maszynowego podawanych procesorowi. Dokładny sposób reprezentacji tych instrukcji maszynowych jest tylko kwestią konwencji i preferencji. Choć teoretycznie można stworzyć własną składnię asemblera dla architektury x86, większość osób wykorzystuje jeden z głównych typów — składnię AT&T lub Intel. Kod asemblera przedstawiony na stronie 21 jest zgodny ze składnią AT&T, ponieważ niemal wszystkie dezasemblery w Linuksie domyślnie wykorzystują tę składnię. Składnię AT&T można łatwo rozpoznać dzięki kakofonii symboli % i $ umieszczanych przed wszystkimi elementami (proszę ponownie spojrzeć na stronę 21). Ten sam kod można wyświetlić w składni Intela, dodając do polecenia `objdump` dodatkową opcję `-M`:

```
reader@hacking:~/booksrc $ objdump -M intel -D a.out | grep -A20 main.:
08048374 <main>:
8048374: 55 push ebp
8048375: 89 e5 mov ebp,esp
```

```
8048377: 83 ec 08 sub esp,0x8
804837a: 83 e4 f0 and esp,0xfffffff0
804837d: b8 00 00 00 00 mov eax,0x0
8048382: 29 c4 sub esp,eax
8048384: c7 45 fc 00 00 00 00 mov DWORD PTR [ebp-4],0x0
804838b: 83 7d fc 09 cmp DWORD PTR [ebp-4],0x9
804838f: 7e 02 jle 8048393 <main+0x1f>
8048391: eb 13 jmp 80483a6 <main+0x32>
8048393: c7 04 24 84 84 04 08 mov DWORD PTR [esp],0x8048484
804839a: e8 01 ff ff ff call 80482a0 <printf@plt>
804839f: 8d 45 fc lea eax,[ebp-4]
80483a2: ff 00 inc DWORD PTR [eax]
80483a4: eb e5 jmp 804838b <main+0x17>
80483a6: c9 leave
80483a7: c3 ret
80483a8: 90 nop
80483a9: 90 nop
80483aa: 90 nop
reader@hacking:~/booksrc $
```

Uważam, że składnia Intela jest znacznie bardziej czytelna i łatwiejsza do zrozumienia, więc będzie ona stosowana w niniejszej książce. Niezależnie od reprezentacji języka asemblera, polecenia rozumiane przez procesor są całkiem proste. Instrukcje te składają się z operacji i czasem dodatkowych operacji opisujących cel i (lub) źródło operacji. Operacje te przenoszą dane w pamięci, aby wykonać jakiegoś rodzaju podstawowe działania matematyczne, lub przerywają działanie procesora, by wykonać coś innego. To jest wszystko, co procesor potrafi wykonać. Jednak, podobnie jak za pomocą stosunkowo niewielkiego alfabetu napisano miliony książek, tak samo, korzystając z niewielkiego zbioru instrukcji maszynowych, można utworzyć nieskończenie wiele programów.

Procesory posiadają również własny zestaw specjalnych zmiennych zwanych *rejestrami*. Większość instrukcji wykorzystuje rejestry do odczytywania lub zapisywania danych, więc ich poznanie jest niezbędne do zrozumienia instrukcji. Duży obraz wciąż się powiększa…

0x252 Procesor x86

Procesor 8086 był pierwszym procesorem w architekturze *x86*. Został opracowany i wyprodukowany przez firmę Intel, która później wprowadzała na rynek bardziej zaawansowane procesory z tej rodziny: 80186, 80286m 80386 i 80486. Jeżeli pamiętasz, że w latach 80. i 90. ubiegłego wieku mówiono o procesorach 386 i 486, chodziło wówczas o te właśnie układy.

Procesor x86 posiada kilka rejestrów, które są dla niego czymś w rodzaju wewnętrznych zmiennych. Mógłbym teraz przeprowadzić abstrakcyjny wykład na temat rejestrów, ale uważam, że lepiej zobaczyć coś na własne oczy. Narzędzia programistyczne GNU zawierają również debuger o nazwie GDB. *Debugery* są używane przez programistów do wykonywania skompilowanych programów krok po kroku, badania pamięci programu i przeglądania rejestrów procesora. Programista, który

nigdy nie korzystał z debugera do przyjrzenia się wewnętrznym mechanizmom pracy programu, jest jak siedemnastowieczny badacz, który nigdy nie używał mikroskopu. Debuger, tak jak mikroskop, pozwala hakerowi na precyzyjne przyjrzenie się kodowi maszynowemu, ale możliwości debugera wykraczają daleko poza tę metaforę. W przeciwieństwie do mikroskopu, debuger umożliwia obejrzenie wykonywania programu ze wszystkich stron, wstrzymanie go oraz zmianę wszelkich parametrów.

Poniżej GDB został użyty do wyświetlenia stanu rejestrów procesora tuż przed rozpoczęciem programu:

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main

Breakpoint 1 at 0x804837a
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, 0x0804837a in main ()
(gdb) info registers
eax 0xbffff894 -1073743724
ecx 0x48e0fe81 1222704769
edx 0x1 1
ebx 0xb7fd6ff4 -1208127500
esp 0xbffff800 0xbffff800
ebp 0xbffff808 0xbffff808
esi 0xb8000ce0 -1207956256
edi 0x0 0
eip 0x804837a 0x804837a <main+6>
eflags 0x286 [ PF SF IF ]
cs 0x73 115
ss 0x7b 123
ds 0x7b 123
es 0x7b 123
fs 0x0 0
gs 0x33 51
(gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

Punkt wstrzymania został ustawiony przy funkcji `main()`, więc wykonywanie programu zostanie zatrzymane tuż przed uruchomieniem naszego kodu. Następnie GDB uruchamia program, zatrzymuje się w punkcie wstrzymania i wyświetla wszystkie rejestry procesora i ich bieżący stan.

Pierwsze cztery rejestry (EAX, ECX, EDX i EBX) są rejestrami ogólnego przeznaczenia. Są to odpowiednio rejestry: *akumulatorowy*, *licznika*, *danych* i *bazowy*. Są one wykorzystywane do różnych celów, ale służą głównie jako zmienne tymczasowe dla procesora, gdy wykonuje instrukcje kodu maszynowego.

Kolejne cztery rejestry (ESP, EBP, ESI i EDI) są również rejestrami ogólnego przeznaczenia, ale czasem nazywa się je wskaźnikami i indeksami. Są to odpowiednio: *wskaźnik*, *wskaźnik bazowy*, *źródło* i *przeznaczenie*. Pierwsze dwa rejestry są nazywane wskaźnikami, ponieważ przechowują 32-bitowe adresy, wskazujące do miejsc w pamięci. Te rejestry są istotne dla wykonywania programu i zarządzania pamięcią. Później omówię je dokładniej. Kolejne dwa rejestry, z technicznego punktu widzenia również wskaźnikowe, często wykorzystywane są do wskazywania źródła i celu, gdy dane muszą być odczytane lub zapisane. Istnieją instrukcje odczytujące i zapisujące, które używają tych rejestrów, ale przeważnie można je traktować jak zwykłe rejestry ogólnego przeznaczenia.

Rejestr *EIP* jest rejestrem *wskaźnika instrukcji*, wskazującym aktualnie wykonywaną instrukcję. Tak jak dziecko podczas czytania pokazuje sobie palcem poszczególne słowa, tak procesor odczytuje konkretne instrukcje, korzystając z EIP jak z palca. Oczywiście, rejestr ten jest bardzo istotny i będzie często wykorzystywany podczas debugowania. Aktualnie wskazuje adres pamięci 0x804838a.

Pozostały rejestr, *EFLAGS*, zawiera kilka flag bitowych wykorzystywanych do porównań oraz segmentacji pamięci. Pamięć jest dzielona na kilka różnych segmentów, co opiszę później, a rejestr ten pozwala je śledzić. Przeważnie możemy je zignorować, ponieważ rzadko konieczny jest bezpośredni dostęp do nich.

0x253 Język asembler

Ponieważ w niniejszej książce wykorzystujemy składnię Intela w języku asembler, musimy odpowiednio skonfigurować narzędzia. W GDB składnię dezasemblera można ustawić na intelowską, wpisując po prostu `set disassembly intel` lub w skrócie `set dis intel`. Możemy skonfigurować GDB tak, aby to ustawienie było aktywne przy każdym uruchomieniu, umieszczając to polecenie w pliku *.gdbinit* w katalogu domowym.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) set dis intel
(gdb) quit
reader@hacking:~/booksrc $ echo "set dis intel" > ~/.gdbinit
reader@hacking:~/booksrc $ cat ~/.gdbinit
set dis intel
reader@hacking:~/booksrc $
```

Po skonfigurowaniu GDB tak, żeby wykorzystywał składnię Intela, zapoznajmy się z nią. W składni Intela instrukcje asemblera mają zwykle poniższą postać:

```
operacja <cel>, <źródło>
```

Wartościami docelowymi i źródłowymi są rejestr, adres pamięci lub wartość. Operacje są zwykle intuicyjnie rozumianymi mnemonikami. Operacja `mov` przenosi wartość ze źródła do celu, `sub` odejmuje, `inc` inkrementuje itd. Przykładowo poniższe instrukcje przenoszą wartość z ESP do EBP, a następnie odejmują 8 od ESP (zapisując wynik w ESP).

```
8048375: 89 e5 mov ebp,esp
8048377: 83 ec 08 sub esp,0x8
```

Dostępne są również operacje wykorzystywane do sterowanie przebiegiem wykonywania. Operacja `cmp` służy do porównywania wartości, a niemal wszystkie operacje, których nazwa rozpoczyna się literą `j`, służą do przeskakiwania do innej części kodu (w zależności od wyniku porównania). W poniższym przykładzie najpierw 4-bajtowa wartość znajdująca w ERB minus 4 jest porównywana z liczbą 9. Następna instrukcja jest skrótem od *przeskocz, jeżeli mniejsze lub równe niż* (ang. *jump if less than or equal to*), odnoszącym się do wyniku wcześniejszego porównania. Jeżeli wartość ta jest mniejsza lub równa 9, wykonywanie przeskoczy do instrukcji znajdującej się pod adresem 0x8048393. W przeciwnym przypadku wykonywana będzie kolejna instrukcja z bezwarunkowym przeskokiem. Jeżeli wartość nie będzie mniejsza ani równa 9, procesor przeskoczy do 0x80483a6.

```
804838b: 83 7d fc 09 cmp DWORD PTR [ebp-4],0x9
804838f: 7e 02 jle 8048393 <main+0x1f>
8048391: eb 13 jmp 80483a6 <main+0x32>
```

Przykłady te pochodzą z naszej poprzedniej dezasemblacji. Skonfigurowaliśmy debuger do wykorzystywania składni Intela, więc wykorzystajmy go do kroczenia przez nasz pierwszy program na poziomie instrukcji asemblera.

W kompilatorze GCC można podać opcję `-g`, dzięki czemu zostaną dołączone dodatkowe informacje debugowania, co da GDB dostęp do kodu źródłowego.

```
reader@hacking:~/booksrc $ gcc -g firstprog.c
reader@hacking:~/booksrc $ ls -l a.out
-rwxr-xr-x 1 matrix users 11977 Jul 4 17:29 a.out
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) list
1 #include <stdio.h>
2
3 int main()
4 {
5 int i;
6 for(i=0; i < 10; i++)
7 {
8 printf("Hello, world!\n");
9 }
10 }
(gdb) disassemble main
Dump of assembler code for function main():
0x08048384 <main+0>: push ebp
0x08048385 <main+1>: mov ebp,esp
0x08048387 <main+3>: sub esp,0x8
0x0804838a <main+6>: and esp,0xfffffff0
0x0804838d <main+9>: mov eax,0x0
0x08048392 <main+14>: sub esp,eax
```

```
0x08048394 <main+16>: mov DWORD PTR [ebp-4],0x0
0x0804839b <main+23>: cmp DWORD PTR [ebp-4],0x9
0x0804839f <main+27>: jle 0x80483a3 <main+31>
0x080483a1 <main+29>: jmp 0x80483b6 <main+50>
0x080483a3 <main+31>: mov DWORD PTR [esp],0x80484d4
0x080483aa <main+38>: call 0x80482a8 <_init+56>
0x080483af <main+43>: lea eax,[ebp-4]
0x080483b2 <main+46>: inc DWORD PTR [eax]
0x080483b4 <main+48>: jmp 0x804839b <main+23>
0x080483b6 <main+50>: leave
0x080483b7 <main+51>: ret
End of assembler dump.
(gdb) break main
Breakpoint 1 at 0x8048394: file firstprog.c, line 6.
(gdb) run
Starting program: /hacking/a.out

Breakpoint 1, main() at firstprog.c:6
6 for(i=0; i < 10; i++)
(gdb) info register eip
eip 0x8048394 0x8048394
(gdb)
```

Najpierw wypisywany jest kod źródłowy oraz dezasemblacja funkcji `main()`. Następnie na początku `main()` ustawiany jest punkt wstrzymania i program jest uruchamiany. Ten punkt wstrzymania po prostu informuje debuger, aby wstrzymał wykonywanie programu, gdy dojdzie do tego punktu. Ponieważ punkt wstrzymania został ustawiony na początku funkcji `main()`, program dociera do punktu wstrzymania i zatrzymuje się przed wykonaniem jakichkolwiek instrukcji z funkcji `main()`. Następnie wyświetlana jest wartość EIP (wskaźnika instrukcji).

Należy zwrócić uwagę, że EIP zawiera adres pamięci, wskazujący instrukcję w dezasemblacji funkcji `main()` (pogrubionej na listingu). Wcześniejsze instrukcje (zapisane kursywą), razem nazywane *prologiem funkcji*, są generowane przez kompilator w celu ustawienia pamięci dla reszty zmiennych lokalnych funkcji `main()`. Jednym z powodów konieczności deklaracji zmiennych w języku C jest pomoc w konstrukcji tej części kodu. Debuger „wie", że ta część kodu jest generowana automatycznie i potrafi ją pominąć. Prologiem funkcji zajmiemy się później, a na razie, wzorując się na GDB, pominiemy go.

Debuger GDB zawiera bezpośrednią metodę badania pamięci za pomocą polecenia `x`, które jest skrótem od *examine*. Badanie pamięci jest bardzo ważną umiejętnością każdego hakera. Większość technik hakerskich przypomina sztuczki magiczne — wydają się niesamowite i magiczne, dopóki nie poznasz oszustwa. Zarówno w magii, jak i hakerstwie, jeżeli popatrzymy w odpowiednie miejsce, sztuczka stanie się oczywista. Jest to jeden z powodów, dla których dobry magik nigdy nie wykonuje dwa razy tego samego triku w czasie jednego występu. Jednak za pomocą debugera, takiego jak GDB, możemy dokładnie zbadać każdy aspekt wykonywania programu, wstrzymać go, przejść krok dalej i powtórzyć tyle razy, ile potrzeba. Ponieważ działający program to w większości procesor i segmenty pamięci, zbadanie pamięci jest pierwszą z metod przekonania się, co naprawdę się dzieje.

Polecenie `examine` w GDB może zostać użyte do przyjrzenia się określonym adresom pamięci na różne sposoby. To polecenie oczekuje dwóch argumentów: miejsca w pamięci, które ma być zbadane, oraz określenia sposobu wyświetlania zawartości.

Do określania formatu wyświetlania również używany jest jednoliterowy skrót, który można poprzedzić liczbą elementów do zbadania. Najczęściej wykorzystywane są poniższe skróty formatujące:

- **o** — wyświetla w formacie ósemkowym,
- **x** — wyświetla w formacie szesnastkowym,
- **u** — wyświetla w standardowym systemie dziesiętnym, bez znaków,
- **t** — wyświetla w formacie binarnym.

Liter tych można użyć z poleceniem `examine` do zbadania określonego adresu pamięci. W poniższym przykładzie wykorzystano bieżący adres z rejestru EIP. Skrócone polecenia są często wykorzystywane w GDB i nawet `info register eip` można skrócić do `i r eip`.

```
(gdb) i r eip
eip            0x8048384        0x8048384 <main+16>
(gdb) x/o 0x8048384
0x8048384 <main+16>:    077042707
(gdb) x/x $eip
0x8048384 <main+16>:    0x00fc45c7
(gdb) x/u $eip
0x8048384 <main+16>:    16532935
(gdb) x/t $eip
0x8048384 <main+16>:    00000000111111000100010111000111
(gdb)
```

Pamięć, którą wskazuje rejestr EIP, może być zbadana za pomocą adresu przechowywanego w EIP. Debuger umożliwia bezpośrednie odwoływanie się do rejestrów, więc `$eip` jest równoważne wartości przechowywanej aktualnie w EIP. Wartość 077042707 w systemie ósemkowym jest tym samym, co 0x00fc45c7 w systemie szesnastkowym, co jest tym samym, co 16532935 w systemie dziesiętnym, co z kolei jest tym samym, co 00000000111111000100010111000111 w systemie dwójkowym. Format polecenia `examine` można również poprzedzić cyfrą, określającą liczbę badanych jednostek w docelowym adresie.

```
(gdb) x/2x $eip
0x8048384 <main+16>: 0x00fc45c7 0x83000000
(gdb) x/12x $eip
0x8048384 <main+16>: 0x00fc45c7 0x83000000 0x7e09fc7d 0xc713eb02
0x8048394 <main+32>: 0x84842404 0x01e80804 0x8dffffff 0x00fffc45
0x80483a4 <main+48>: 0xc3c9e5eb 0x90909090 0x90909090 0x5de58955
(gdb)
```

Domyślnym rozmiarem jednej jednostki jest 4-bajtowa jednostka zwana *słowem*. Rozmiar jednostek wyświetlania dla polecenia `examine` może być zmieniany poprzez dodanie odpowiedniej litery za literą formatującą. Poprawnymi literami rozmiaru są:

b — pojedynczy bajt,

h — półsłowo o rozmiarze dwóch bajtów,

w — słowo o rozmiarze czterech bajtów,

g — słowo podwójne o rozmiarze ośmiu bajtów.

Wprowadza to pewien zamęt, ponieważ czasami termin *słowo* określa również wartości 2-bajtowe. W takim przypadku *podwójne słowo* lub *DWORD* oznacza wartość 4-bajtową. W książce tej terminy „słowo" i DWORD będą określały wartości 4-bajtowe. Do określania wartości 2-bajtowych będzie używany termin „półsłowo". Poniższe wyjście z GDB obrazuje pamięć wyświetlaną w różnych rozmiarach.

```
(gdb) x/8xb $eip
0x8048384 <main+16>: 0xc7   0x45   0xfc   0x00   0x00   0x00   0x00   0x83
(gdb) x/8xh $eip
0x8048384 <main+16>: 0x45c7 0x00fc 0x0000 0x8300 0xfc7d 0x7e09 0xeb02 0xc713
(gdb) x/8xw $eip
0x8048384 <main+16>: 0x00fc45c7    0x83000000    0x7e09fc7d    0xc713eb02
0x8048394 <main+32>: 0x84842404    0x01e80804    0x8dffffff    0x00fffc45
(gdb)
```

Jeżeli dobrze się przyjrzysz, zauważysz w powyższych danych coś dziwnego. Pierwsze polecenie `examine` pokazuje pierwsze osiem bajtów i, oczywiście, polecenia `examine` wykorzystujące większe jednostki wyświetlają więcej danych. Jednakże wynik pierwszego polecenia pokazuje, że pierwszymi dwoma bajtami są 0xc7 i 0x45, ale gdy badane jest półsłowo w dokładnie tym samym adresie pamięci, pokazywana jest wartość 0x45c7, czyli bajty są odwrócone. Ten sam efekt możemy zauważyć w przypadku pełnego 4-bajtowego słowa wyświetlanego jako 0x00fc45c7. Jeżeli jednak pierwsze cztery bajty są wyświetlane bajt po bajcie mają one kolejność 0xc7, 0x45m 0xfc i 0x00.

Jest tak, ponieważ w procesorze x86 wartości są przechowywane w kolejności *little endian*, co oznacza, że najmniej istotny bajt jest przechowywany jako pierwszy. Jeżeli np. cztery bajty mają być interpretowane jako jedna wartość, bajty muszą być użyte w odwrotnej kolejności. Debuger GDB jest na tyle mądry, że wie, jak przechowywane są wartości, więc gdy badane jest słowo lub półsłowo, bajty muszą być odwrócone w celu wyświetlenia poprawnych wartości w układzie szesnastkowym. Przejrzenie tych wyświetlanych wartości w systemie szesnastkowym i jako liczb dziesiętnych bez znaku może ułatwić zrozumienie tej kwestii.

```
(gdb) x/4xb $eip
0x8048384 <main+16>:   0xc7  0x45  0xfc  0x00
(gdb) x/4ub $eip
0x8048384 <main+16>:   199   69    252   0
(gdb) x/1xw $eip
```

```
0x8048384 <main+16>:   0x00fc45c7
(gdb) x/1uw $eip
0x8048384 <main+16>:   16532935
(gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $ bc -ql
199*(256^3) + 69*(256^2) + 252*(256^1) + 0*(256^0)
3343252480
0*(256^3) + 252*(256^2) + 69*(256^1) + 199*(256^0)
16532935
quit
reader@hacking:~/booksrc $
```

Pierwsze cztery bajty są pokazane zarówno w notacji szesnastkowej, jak i standardowej, dziesiętnej. Kalkulator działający w wierszu poleceń `bc` posłużył do wykazania, że jeżeli bajty zostaną zinterpretowane w nieprawidłowej kolejności, otrzymamy w wyniku zupełnie nieprawidłową wartość 3343252480. Kolejność bajtów w danej architekturze jest istotnym czynnikiem, którego należy być świadomym. Choć większość narzędzi debugujących i kompilatorów zajmuje się automatycznie szczegółami związanymi z kolejnością bajtów, czasem będziesz musiał samodzielnie manipulować pamięcią.

Oprócz konwersji kolejności bajtów, GDB może za pomocą polecenia `examine` wykonywać inne konwersje. Widzieliśmy już, że GDB może dezasemblować instrukcje języka maszynowego na zrozumiałe dla człowieka instrukcje asemblera. Polecenie `examine` przyjmuje również literę formatującą `i` (skrót od *instruction*), co powoduje wyświetlenie pamięci jako dezasemblowanych instrukcji języka asembler.

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x8048384: file firstprog.c, line 6.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at firstprog.c:6
6         for(i=0; i < 10; i++)
(gdb) i r $eip
eip            0x8048384       0x8048384 <main+16>
(gdb) x/i $eip
0x8048384 <main+16>:  mov   DWORD PTR [ebp-4],0x0
(gdb) x/3i $eip
0x8048384 <main+16>:  mov   DWORD PTR [ebp-4],0x0
0x804838b <main+23>:  cmp   DWORD PTR [ebp-4],0x9
0x804838f <main+27>:  jle   0x8048393 <main+31>
(gdb) x/7xb $eip
0x8048384 <main+16>:  0xc7  0x45  0xfc  0x00  0x00  0x00  0x00
(gdb) x/i $eip
0x8048384 <main+16>:  mov   DWORD PTR [ebp-4],0x0
(gdb)
```

W powyższym przykładzie program *a.out* został uruchomiony w GDB z punktem wstrzymania ustawionym w `main()`. Ponieważ rejestr EIP wskazuje na pamięć, która faktycznie zawiera instrukcje w języku maszynowym, ładnie poddają się one procesowi dezasemblacji.

Wcześniejsza dezasemblacja za pomocą `objectdump` potwierdza, że siedem bajtów, które wskazuje EIP, to faktycznie język maszynowy dla odpowiedniej instrukcji asemblera.

```
8048384:   c7 45 fc 00 00 00 00   mov   DWORD PTR [ebp-4],0x0
```

Ta instrukcja asemblera przenosi wartość 0 do pamięci znajdującej się pod adresem przechowywanym w rejestrze EBP minus 4. To właśnie w tym miejscu pamięci przechowywana jest zmienna `i` z języka C. Zmienna ta została zadeklarowana jako liczba całkowita (`int`), wykorzystująca 4 bajty pamięci w przypadku procesora x86. To polecenie po prostu zeruje zmienną `i` dla pętli `for`. Gdybyśmy teraz zbadali tę pamięć, zawierałaby ona tylko przypadkowe śmieci. Pamięć w tej lokacji można zbadać na różne sposoby:

```
(gdb) i r ebp
ebp            0xbffff808      0xbffff808
(gdb) x/4xb $ebp - 4
0xbffff804:     0xc0    0x83   0x04   0x08
(gdb) x/4xb 0xbffff804
0xbffff804:     0xc0    0x83   0x04   0x08
(gdb) print $ebp - 4
$1 = (void *) 0xbffff804
(gdb) x/4xb $1
0xbffff804:     0xc0    0x83   0x04   0x08
(gdb) x/xw $1
0xbffff804:     0x080483c0
(gdb)
```

Rejestr EBP zawiera adres 0xbffff808, a instrukcja asemblera będzie zapisywała do adresu o tej wartości pomniejszonej o 4 — 0xbffff804. Polecenie `examine` może zbadać bezpośrednio ten adres pamięci lub wykonać odpowiednie obliczenia w locie. Za pomocą polecenia `print` również można wykonać proste działania arytmetyczne, a wynik zostanie zapisany w tymczasowej zmiennej w debugerze. Zmienna ta nosi nazwę `$1` i może później zostać użyta do szybkiego uzyskania dostępu do określonego miejsca w pamięci. Każda z przedstawionych wyżej metod pozwala uzyskać ten sam rezultat: wyświetlenie znalezionych w pamięci 4 bajtów śmieci, które zostaną wyzerowane po wykonaniu bieżącej instrukcji.

Uruchommy bieżącą instrukcję, korzystając z polecenia `nexti`, będącego skrótem dla *next instruction* (następna instrukcja). Procesor odczyta instrukcję z EIP, wykona ją i przestawi EIP na kolejną instrukcję.

```
(gdb) nexti
0x0804838b     6        for(i=0; i < 10; i++)
(gdb) x/4xb $1
0xbffff804:    0x00   0x00   0x00   0x00
(gdb) x/dw $1
0xbffff804:    0
(gdb) i r eip
eip            0x804838b      0x804838b <main+23>
(gdb) x/i $eip
0x804838b <main+23>:   cmp DWORD PTR [ebp-4],0x9
(gdb)
```

Zgodnie z oczekiwaniami, poprzednia instrukcja zeruje 4 bajty odnalezione w adresie EBP minus 4, czyli pamięci przydzielonej zmiennej `i` z kodu C. EIP przechodzi do kolejnej instrukcji. Omówienie kolejnych instrukcji warto przeprowadzić łącznie.

```
(gdb) x/10i $eip
0x804838b <main+23>: cmp   DWORD PTR [ebp-4],0x9
0x804838f <main+27>: jle   0x8048393 <main+31>
0x8048391 <main+29>: jmp   0x80483a6 <main+50>
0x8048393 <main+31>: mov   DWORD PTR [esp],0x8048484
0x804839a <main+38>: call  0x80482a0 <printf@plt>
0x804839f <main+43>: lea   eax,[ebp-4]
0x80483a2 <main+46>: inc   DWORD PTR [eax]
0x80483a4 <main+48>: jmp   0x804838b <main+23>
0x80483a6 <main+50>: leave
0x80483a7 <main+51>: ret
(gdb)
```

Pierwsza instrukcja, `cmp`, jest instrukcją porównania, która porównuje zawartość pamięci wykorzystywanej przez zmienną i z kodu C z wartością 9. Kolejna instrukcja, `jle`, oznacza *przeskocz, jeżeli mniejsze lub równe* (ang. *jump if less than equal to*). Wykorzystuje ona wynik poprzedniego porównania (który jest przechowywany w rejestrze EFLAGS) do przestawienia EIP tak, aby wskazywał inną część kodu, jeżeli cel poprzedniej operacji porównania jest mniejszy lub równy źródłu. W tym przypadku instrukcja powoduje przeskok do adresu 0x8048393, jeżeli wartość przechowywana w pamięci dla zmiennej `i` jest mniejsza lub równa wartości 9. W przeciwnym przypadku EIP przejdzie do kolejnej instrukcji, która jest instrukcją przeskoku bezwarunkowego. Powoduje ona przeskok EIP do adresu 0x80483a6. Te trzy instrukcje łącznie tworzą strukturę sterującą `if-then-else`: *jeżeli `i` jest mniejsze lub równe 9, przejdź do instrukcji pod adresem 0x8048393; w przeciwnym przypadku przejdź do instrukcji pod adresem 0x80483a6*. Pierwszy adres, 0x804893 (pogrubiony), jest po prostu stałą instrukcją przeskoku, a drugi adres — 0x80483a6 (zapisany kursywą) — znajduje się na końcu funkcji.

Ponieważ wiemy, że w lokacji pamięci porównywanej z 9 przechowywane jest 0 i wiemy, że 0 jest mniejsze lub równe 9, po wykonaniu kolejnych dwóch instrukcji EIP powinien wskazywać na 0x8048393.

```
(gdb) nexti
0x0804838f       6          for(i=0; i < 10; i++)
(gdb) x/i $eip
0x804838f <main+27>:      jle    0x8048393 <main+31>
(gdb) nexti
8          printf("Hello world!\n");
(gdb) i r eip
eip            0x8048393         0x8048393 <main+31>
(gdb) x/2i $eip
0x8048393 <main+31>:      mov    DWORD PTR [esp],0x8048484
0x804839a <main+38>:      call   0x80482a0 <printf@plt>
(gdb)
```

Zgodnie z oczekiwaniami, wcześniejsze dwie instrukcje przeniosły wykonywanie programu do 0x8048393, co doprowadza do kolejnych dwóch instrukcji. Pierwsza z nich jest kolejną instrukcją `mov`, która zapisze adres 0x8048484 do adresu pamięci znajdującego się w rejestrze ESP. Ale co wskazuje ESP?

```
(gdb) i r esp
esp           0xbffff800 0xbffff800
(gdb)
```

W tej chwili ESP wskazuje adres pamięci 0xbffff800, więc po wykonaniu instrukcji `mov` zostanie tutaj zapisany adres 0x8048484. Ale dlaczego? Co takiego specjalnego jest w adresie pamięci 0x8048484? Jest tylko jeden sposób, aby się o tym przekonać.

```
(gdb) x/2xw 0x8048484
0x8048484:      0x6c6c6548   0x6f57206f
(gdb) x/6xb 0x8048484
0x8048484:      0x48    0x65 0x6c    0x6c 0x6f 0x20
(gdb) x/6ub 0x8048484
0x8048484:      72      101  108     108  111  32
(gdb)
```

Doświadczony czytelnik może zauważyć w tej pamięci coś szczególnego, zwłaszcza w zakresie bajtów. Po nabraniu doświadczenia w badaniu pamięci tego typu wzorce staną się bardziej oczywiste. Bajty te zawierają się w zakresie znaków ASCII. *ASCII* jest uzgodnionym standardem, który odwzorowuje wszystkie znaki z klawiatury (a także kilka innych) na stałe liczby. Bajty 0x48, 0x65 i 0x6f odpowiadają literom alfabetu przedstawionym w tabeli znaków ASCII (tabela 2.1). Tabelę można znaleźć na stronach podręcznika ASCII dostępnego w większości systemów Unix po wpisaniu `man ascii`.

Na szczęście, polecenie `examine` w GDB posiada również możliwość przyjrzenia się tego typu pamięci. Litera formatująca `c` umożliwia automatyczne sprawdzenie bajta w tablicy ASCII, a litera formatująca `s` powoduje wyświetlenie całego łańcucha danych znakowych.

Tabela 2.1. *Tablica znaków ASCII*

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL '\0'	100	64	40	@
001	1	01	SOH	101	65	41	A
002	2	02	STX	102	66	42	B
003	3	03	ETX	103	67	43	C
004	4	04	EOT	104	68	44	D
005	5	05	ENQ	105	69	45	E
006	6	06	ACK	106	70	46	F
007	7	07	BEL '\a'	107	71	47	G
010	8	08	BS '\b'	110	72	48	H
011	9	09	HT '\t'	111	73	49	I
012	10	0A	LF '\n'	112	74	4A	J
013	11	0B	VT '\v'	113	75	4B	K
014	12	0C	FF '\f'	114	76	4C	L
015	13	0D	CR '\r'	115	77	4D	M
016	14	0E	SO	116	78	4E	N
017	15	0F	SI	117	79	4F	O
020	16	10	DLE	120	80	50	P
021	17	11	DC1	121	81	51	Q
022	18	12	DC2	122	82	52	R
023	19	13	DC3	123	83	53	S
024	20	14	DC4	124	84	54	T
025	21	15	NAK	125	85	55	U
026	22	16	SYN	126	86	56	V
027	23	17	ETB	127	87	57	W
030	24	18	CAN	130	88	58	X
031	25	19	EM	131	89	59	Y
032	26	1A	SUB	132	90	5A	Z
033	27	1B	ESC	133	91	5B	[
034	28	1C	FS	134	92	5C	\ '\\'
035	29	1D	GS	135	93	5D	]
036	30	1E	RS	136	94	5E	^
037	31	1F	US	137	95	5F	_
040	32	20	SPACE	140	96	60	
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c

Tabela 2.1. *Tablica znaków ASCII — ciąg dalszy*

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
044	36	24	$	144	100	64	d
045	37	25	%	**145**	**101**	**65**	**e**
046	38	26	&	146	102	66	f
047	39	27	'	147	103	67	g
050	40	28	(	150	104	68	h
051	41	29	)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	**154**	**108**	**6C**	**l**
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	**157**	**111**	**6F**	**o**
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	|
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

```
(gdb) x/6cb 0x8048484
0x8048484:      72 'H' 101 'e' 108 'l' 108 'l' 111 'o' 32 ' '
(gdb) x/s 0x8048484
0x8048484:      "Hello, world!\n"
(gdb)
```

Polecenia te pokazują, że pod adresem 0x8048484 przechowywany jest łańcuch danych `"Hello, world!\n"`. Ten łańcuch jest argumentem funkcji `printf()`, co wskazuje, że przeniesienie adresu tego łańcucha do adresu przechowywanego w ESP

(0x8048484) ma coś wspólnego z tą funkcją. Poniższy wynik z gdb pokazuje adres łańcucha danych przenoszony do adresu, który wskazuje ESP.

```
(gdb) x/2i $eip
0x8048393 <main+31>:    mov     DWORD PTR [esp],0x8048484
0x804839a <main+38>:    call    0x80482a0 <printf@plt>
(gdb) x/xw $esp
0xbffff800:     0xb8000ce0
(gdb) nexti
0x0804839a      8             printf("Hello, world!\n");
(gdb) x/xw $esp
0xbffff800:     0x08048484
(gdb)
```

Kolejna instrukcja wywołuje funkcję printf(). Wypisuje ona łańcuch danych. Wcześniej instrukcje stanowiły przygotowania do wywołania tej funkcji, a wynik tego wywołania został pogrubiony poniżej.

```
(gdb) x/i $eip
0x804839a <main+38>:    call    0x80482a0 <printf@plt>
(gdb) nexti
Hello, world!
6         for(i=0; i < 10; i++)
(gdb)
```

Korzystając dalej z debugera, przyjrzyjmy się kolejnym dwóm instrukcjom. Ponownie większy sens ma opisanie ich razem.

```
(gdb) x/2i $eip
0x804839f <main+43>: lea eax,[ebp-4]
0x80483a2 <main+46>: inc DWORD PTR [eax]
(gdb)
```

Te dwie instrukcje po prostu zwiększają wartość zmiennej i o 1. Instrukcja lea, akronim od *Load Effective Address* (wczytaj efektywny adres), wczytuje znany już adres EBP minus 4 do rejestru EAX. Wykonanie tej instrukcji przedstawiono poniżej.

```
(gdb) x/ i $eip
0x804839f <main+43>:     lea    eax,[ebp-4]
(gdb) print $ebp - 4
$2 = (void *) 0xbffff804
(gdb) x/x $2
0xbffff804:      0x00000000
(gdb) i r eax
eax             0xd      13
(gdb) nexti
0x080483a2       6         for(i=0; i < 10; i++)
(gdb) i r eax
eax             0xbffff804       -1073743868
(gdb) x/xw $eax
```

```
0xbffff804:      0x00000000
(gdb) x/dw $eax
0xbffff804:      0
(gdb)
```

Kolejna instrukcja, `inc`, zwiększy wartość znalezioną pod tym adresem (przechowywaną teraz w rejestrze EAX) o 1. Wykonanie tej instrukcji również przedstawiono poniżej.

```
(gdb) x/i $eip
0x80483a2 <main+46>: inc    DWORD PTR [eax]
(gdb) x/dw $eax
0xbffff804:     0
(gdb) nexti
0x080483a4      6       for(i=0; i < 10; i++)
(gdb) x/dw $eax
0xbffff804:     1
(gdb)
```

W wyniku wartość przechowywana pod adresem pamięci EBP minus 4 (0xbffff804) jest powiększana o 1. To zachowanie odpowiada fragmentowi kodu C, w którym wartość zmiennej `i` jest powiększana w pętli `for`.

Kolejną instrukcją jest bezwarunkowy przeskok.

```
(gdb) x/i $eip
0x80483a4 <main+48>: jmp 0x804838b <main+23>
(gdb)
```

Instrukcja po wykonaniu przekieruje program z powrotem do instrukcji spod adresu 0x804838b. Dokonuje tego poprzez proste ustawienie EIP na tę wartość.

Spoglądając ponownie na pełną dezasemblację, powinieneś stwierdzić, które części kodu C zostały skompilowane do których instrukcji języka maszynowego.

```
(gdb) disass main
Dump of assembler code for function main:
0x08048374 <main+0>:     push   ebp
0x08048375 <main+1>:     mov    ebp,esp
0x08048377 <main+3>:     sub    esp,0x8
0x0804837a <main+6>:     and    esp,0xfffffff0
0x0804837d <main+9>:     mov    eax,0x0
0x08048382 <main+14>:    sub    esp,eax
0x08048384 <main+16>:    mov    DWORD PTR [ebp-4],0x0
0x0804838b <main+23>:    cmp    DWORD PTR [ebp-4],0x9
0x0804838f <main+27>:    jle    0x8048393 <main+31>
0x08048391 <main+29>:    jmp    0x80483a6 <main+50>
0x08048393 <main+31>:    mov    DWORD PTR [esp],0x8048484
0x0804839a <main+38>:    call   0x80482a0 <printf@plt>
0x0804839f <main+43>:    lea    eax,[ebp-4]
0x080483a2 <main+46>:    inc    DWORD PTR [eax]
0x080483a4 <main+48>:    jmp    0x804838b <main+23>
```