Redakcja naukowa
Mateusz Jurczyk i Gynvael Coldwind

PRAKTYCZNA INŻYNIERIA WSTECZNA

METODY, TECHNIKI I NARZĘDZIA

Tomasz Bukowski • Grzegorz Antoniak • Tomasz Kwiecień • Mateusz Krzywicki • Marcin Hartung
Gynvael Coldwind • Hasherezade • Maciej Kotowicz • Michał Kowalczyk
Robert Święcki • Piotr Bania • Mateusz Jurczyk

PWN

Projekt okładki **Sebastian Rosik**

Wydawca **Łukasz Łopuszański**

Redaktorzy prowadzący **Gynvael Coldwind, Mateusz Jurczyk**

Redakcja **Tomasz Łopuszański, Małgorzata Dąbkowska-Kowalik**

Korekta merytoryczna **Mariusz Witkowski, Paweł Zakrzewski, Michał Kowalczyk**

Koordynator produkcji **Anna Bączkowska**

Skład i łamanie **Dariusz Ziach**

ISBN 978-83-01-18951-8

Wydanie I
Warszawa 2016

Wydawnictwo Naukowe PWN SA
02-460 Warszawa, ul. Gottlieba Daimlera 2
tel. 22 69 54 321, faks 22 69 54 288
infolinia 801 33 33 88
e-mail: pwn@pwn.com.pl, reklama@pwn.pl
www.pwn.pl

Druk i oprawa: Drukarnia Grafmar Sp. z o.o.

Spis treści

Wykaz autorów

Tomasz Bukowski (rozdział 1)
Z wykształcenia i pasji – fizyk. Entuzjasta Linuksa, a jego hobby to programowanie. Przygodę z bezpieczeństwem IT rozpoczął jako administrator sieci w akademiku. Pełną „gotowość bojową" osiągnął podczas pracy w CERT Polska, gdzie zajmował się analizą (również wsteczną) złośliwego oprogramowania, eksplorowaniem (a czasem przejmowaniem) botnetów, informatyką śledczą oraz innymi kwestiami związanymi z ITSEC. Obecnie kieruje zespołem Security Threat Intelligence w banku Millennium. Po godzinach lata ze „smokami" w zespole CTF „Dragon Sector".

Grzegorz Antoniak (rozdział 2)
Interesuje się inżynierią oprogramowania w wielu różnych językach oraz inżynierią wsteczną w wielu różnych odmianach: od mechanizmów działania systemu począwszy, poprzez mechanizmy budowy komercyjnych aplikacji oraz ich systemów licencyjnych, analizę zamkniętych formatów binarnych, odzyskiwanie danych, do analizy działania złośliwego oprogramowania i projektowania sposobów jego zwalczania. Od kilku lat pracuje w firmie ESET, próbując zwiększać świadomość o systemach uniksowych w środowiskach Windowsowych, koordynuje także zespół rozszerzający zasięg ochrony przeciwko złośliwemu oprogramowaniu na systemach macOS i Linux.

Tomasz Kwiecień (rozdział 3)
Od młodzieńczych lat pasjonat inżynierii wstecznej, a w szczególności mechanizmów zabezpieczeń. Zainteresowanie metodami analizy złośliwego oprogramowania doprowadziło go w 2010 roku do krakowskiego oddziału firmy ESET, gdzie pracuje na stanowisku Specialized Researcher. Wicekapitan i współzałożyciel drużyny CTF „Delicious Horse". W wolnych chwilach „pochłaniacz" filmów i seriali.

Mateusz Krzywicki (rozdział 4)
Pasjonat inżynierii wstecznej, bezpieczeństwa aplikacji i programowania. Interesuje się technikami deobfuskacji i automatycznej dynamicznej analizy kodu. Przygodę z inżynierią wsteczną zaczynał od analizy działania systemów zabezpieczeń i poznawania wnętrza systemów operacyjnych. Fan skomplikowanych eksploitów wykorzystujących błędy związane z naruszeniem spójności obiektów i struktur w pamięci. Bierze udział w zawodach CTF z zespołem „Delicious Horse", którego jest współzałożycielem. Pracował w krakowskim oddziale firmy ESET, a aktualnie w firmie Microsoft, na stanowisku Security Software Engineer.

Marcin Hartung (rozdział 5)
Absolwent Elektroniki i Telekomunikacji na AGH w Krakowie, gdzie okazjonalnie udziela wykładów o inżynierii wstecznej. Autor kilku publikacji, w tym *Unpack your troubles: .NET packer tricks and countermeasures* zaprezentowanej na konferencji VB2015. Od kilku lat pracuje w firmie ESET jako koordynator zespołu zajmującego się analizą i statycznym rozpakowywaniem archiwów oraz protektorów dla formatów wykonywalnych. Prywatnie – wspinacz i motocyklista.

Gynvael Coldwind (rozdział 6)
Programista–pasjonat z zamiłowaniem do bezpieczeństwa komputerowego i niskopoziomowych aspektów informatyki, a także autor licznych artykułów, publikacji, podcastów oraz wystąpień poświęconych tym tematom. W roku 2013 odebrał w Las Vegas (wspólnie z Mateuszem Jurczykiem) nagrodę Pwnie Award w kategorii „Najbardziej Innowacyjne Badania Naukowe" z dziedziny bezpieczeństwa komputerowego, przyznaną za wspólną pracę pt. *Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns*. Kapitan i współzałożyciel „Dragon Sector" – jednej z najlepszych drużyn Security CTF na świecie. Od 2010 r. mieszka w Zurychu, gdzie pracuje dla firmy Google jako Senior Software Engineer / Information Security Engineer.

Hasherezade (rozdział 7)
Magister inżynier informatyki. Od wieku nastoletniego pasjonuje się programowaniem i inżynierią wsteczną. Aktywnie bierze udział w życiu internetowej społeczności InfoSec, dzieląc się efektami swojej pracy – tworząc m.in. programy freeware/open source oraz publikując artykuły techniczne. Obecnie pracuje jako analityk złośliwego oprogramowania dla firmy Malwarebytes; prowadzi także własną działalność w branży.

Maciej Kotowicz (rozdział 8)
Miłośnik piwa i tematów pokrewnych. Po godzinach spędza czas w krainie smoków – wraz z resztą drużyny Dragon Sector – biorąc udział (a czasem nawet wygrywając) we wszelkiej maści CTF-ach. Były Wieczny Student i wykładowca w Instytucie Informatyki we Wrocławiu. Bez nagród, ale zdarza mu się występować tu i tam. Na co dzień botnet pwner w CERT Polska, specjalizujący się w analizie złośliwego oprogramowania oraz pisaniu i analizowaniu exploitów.

Michał Kowalczyk (rozdział 9)
Pasjonat inżynierii wstecznej i analizowania rzeczy tylko po to, aby sprawdzić, jak działają. Interesują go niskopoziomowe aspekty działania komputera, takie jak BIOS oraz system operacyjny, jak również mechanizmy ochrony programów przed inżynierią wsteczną oraz kryptografia. Aktywnie bierze udział w zawodach CTF wraz z zespołem „Dragon Sector", którego jest wicekapitanem.

Robert Święcki (rozdział 10)
Badacz problemów z dziedziny bezpieczeństwa komputerowego, szczególnie w zakresie niskopoziomowym oraz systemów operacyjnych. Autor kilku ciekawych narzędzi z tej dziedziny (m.in. fuzzer *honggfuzz*). Członek polskiej drużyny CTF – „Dragon Sector". Nominowany w 2016 roku do prestiżowej nagrody Pwnie Award w kategorii „Best Privilege Escalation Bug" za upublicznienie błędu bezpieczeństwa w firmwarze procesorów firmy AMD. Pracuje w firmie Google na stanowisku Senior Software Engineer / Information Security Engineer.

Piotr Bania (rozdział 11)
Interesuje się programowaniem oraz bezpieczeństwem systemów komputerowych. Jako pierwszy Polak otrzymał nagrodę Pwnie Award w kategorii „Najbardziej Innowacyjne Badania Naukowe" z dziedziny bezpieczeństwa komputerowego. Wykonywał projekty dla Agencji Zaawansowanych Projektów Badawczych w Obszarze Obronności Departamentu Obrony Stanów Zjednoczonych (DARPA, program Cyber Fast Track). Miał okazję demonstrować swoje prace w Pentagonie. Autor publikacji, eksploitów i innych narzędzi, z których najbardziej popularnym stał się program „kon-boot" używany przez organy ścigania, służby specjalne, specjalistów z zakresu informatyki śledczej, a także duże firmy informatyczne z całego świata. Obecnie pracuje w firmie Cisco w zespole Talos, na stanowisku Senior Security Researcher.

Mateusz Jurczyk (rozdział 12)
Wicekapitan i współzałożyciel zespołu „Dragon Sector", drużyny należącej do ścisłej czołówki światowej w zawodach typu Security CTF. Fan przepełnień bufora. Interesuje się bezpieczeństwem systemów i aplikacji klienckich, w szczególności odkrywaniem podatności, ich wykorzystaniem oraz zapobieganiem, ze wskazaniem na jądro systemu Windows. Trzykrotny laureat prestiżowej nagrody Pwnie Award. Występuje na wielu konferencjach dotyczących bezpieczeństwa oprogramowania, takich jak Black Hat, REcon, SyScan, Ruxcon czy 44CON. Na co dzień pracuje dla firmy Google w zespole Project Zero, na stanowisku Senior Software Engineer / Security Engineer.

Wstęp

Inżynieria wsteczna oprogramowania (ang. *reverse code engineering*, w skrócie RE lub RCE) jest procesem analizy budowy i sposobu działania programów komputerowych, zarówno tych przeznaczonych na serwery, komputery osobiste i urządzenia mobilne, jak i sterujących pracą rozmaitych urządzeń przemysłowych, sieciowych czy AGD. Dziedzina ta ma obecnie bardzo wiele zastosowań.

W wielu przypadkach termin „RCE" jest kojarzony przede wszystkim z analizą tzw. złośliwego oprogramowania, potocznie (choć niekoniecznie poprawnie) nazywanego również „wirusami komputerowymi". W pracy tej celem analityka jest stwierdzenie, czy analizowany program faktycznie działa w złej wierze (tj. na niekorzyść użytkownika), a w dalszej kolejności – ustalenie dokładnego przebiegu infekcji oraz skutków zarażenia. To z kolei pozwala na tworzenie sygnatur jednoznacznie identyfikujących dany *malware*, a także narzędzi do jego usunięcia – oba te elementy zwykle wchodzą w skład oprogramowania antywirusowego.

Aby utrudnić pracę analityków, twórcy złośliwego oprogramowania stosują szereg mechanizmów i sztuczek spowalniających analizę – jedną z nich jest oczywiście obfuskacja, która stanowczo zmniejsza czytelność kodu czy wręcz całkowicie ukrywa go za warstwą szyfrowania lub kompresji. Innym często obserwowanym schematem jest wykrywanie faktu dokonywania analizy kodu i jej aktywne utrudnianie np. poprzez przedwczesne zakończenie procesu czy zabicie procesu debuggera. Co ciekawe, dokładnie te same metody są stosowane przez pewną grupę twórców legalnego oprogramowania, którzy starają się chronić w ten sposób zaimplementowane przez siebie mechanizmy, weryfikujące fakt zakupienia przez użytkowników licencji zezwalających na użytkowanie danej aplikacji. Po drugiej stronie barykady w tym przypadku stoją tzw. *crackerzy* oraz piraci komputerowi, którzy eliminują wspomniane mechanizmy i rozpowszechniają programy „za plecami" producenta.

Powiązaną poniekąd, ale jednak odrębną gałęzią inżynierii wstecznej jest modyfikacja istniejącego oprogramowania z zamiarem naniesienia na niego poprawek, rozszerzenia funkcjonalności, przetłumaczenia interfejsu użytkownika czy umożliwienia współpracy z innymi aplikacjami. Przykładem takiego działania może być zmiana stawki podatku VAT w archaicznym systemie wspomagającym sprzedaż w firmie, którego producent już dawno nie istnieje. Za inne przykłady mogą posłużyć: zautomatyzowanie konwersji bazy

danych zapisanej w nieznanym formacie do formatu obsługiwanego przez inne aplikacje, poprawienie błędów uniemożliwiających stabilną pracę aplikacji na nowszej wersji systemu operacyjnego czy też dodanie nowych modułów rozbudowujących lub modyfikujących gry komputerowe.

Wreszcie inżynieria wsteczna jest również stosowana w branży bezpieczeństwa komputerowego, a konkretniej w analizie poprawności implementacji (czyli tzw. *vulnerability research*, *bug hunting*), której celem jest poznanie wybranych komponentów oprogramowania na tyle dobrze, aby dogłębnie przebadać je pod kątem występowania błędów bezpieczeństwa. Tak wnikliwa analiza kodu wykonywalnego bardzo często prowadzi do sytuacji, w której badacz zna jego niektóre fragmenty lepiej niż sam twórca.

Można by powiedzieć, że inżynieria wsteczna jest w pewnym stopniu częścią normalnego procesu programowania, gdyż nawet mając dostęp do kodu źródłowego i kompletnej dokumentacji, programiści nierzadko muszą poświęcić czas i energię, aby przeanalizować i zrozumieć nieznane sobie, istotne fragmenty projektu czy nawet własnego kodu, którego szczegóły zdążyły umknąć z pamięci.

Podsumowując: inżynieria wsteczna jest z pewnością fascynującą dziedziną informatyki, w której można znaleźć wiele ciekawych technologii, podejść, trików i niuansów. Z tym większą dumą oddajemy w Państwa ręce niniejszy zbiór publikacji dwunastu autorów – specjalistów z zakresu inżynierii wstecznej z wieloletnim stażem, którzy na co dzień zajmują się analizą złośliwego oprogramowania i badaniem bezpieczeństwa aplikacji, a także biorą udział w turniejach security CTF jako członkowie najlepszych drużyn na świecie.

W tomie tym zebrano materiały dotyczące inżynierii wstecznej przeznaczone zarówno dla początkujących, jak i bardziej zaawansowanych czytelników. Tych pierwszych zainteresują tematy takie jak metody czytania niskopoziomowego kodu, omówienie interesujących aspektów formatów plików wykonywalnych czy charakterystycznych wzorców wynikających z dodawanych przez współczesne kompilatory zabezpieczeń. Osoby od dłuższego czasu związane z tą dziedziną znajdą tu zarówno rozdziały, których tematyka wykracza poza typowy schemat analizy kodu na architekturę x86 – takie jak ciekawostki związane z inżynierią wsteczną mniej standardowych technologii (CPython, .NET), jak i rozdziały zagłębiające się w dobrze znane zagadnienia: śledzenie toku wykonania kodu, sztuczki w formacie ELF, przekierowania API w formacie PE czy techniki związane ze wstrzykiwaniem kodu w inne procesy w systemie Windows. Ostatnie dwa rozdziały książki to studia przypadków, w których omówiono analizę i wykorzystanie niskopoziomowych błędów bezpieczeństwa – najpierw w oprogramowaniu sterującym pracą routera, a następnie w wielu zróżnicowanych zadaniach z turniejów CTF, stworzonych przez autora opracowania.

Życzymy przyjemnej i przede wszystkim interesującej lektury!

Gynvael Coldwind, Mateusz Jurczyk
Wrzesień 2016

Podziękowania

Do powstania niniejszej książki przyczyniło się wiele osób, którym chcielibyśmy w tym miejscu podziękować: Łukasz Łopuszański (wydawca), Tomasz Łopuszański (redaktor), Małgorzata Dąbkowska-Kowalik (redaktor), Mariusz „maryush" Witkowski (recenzent merytoryczny), Paweł „KrzaQ" Zakrzewski (recenzent merytoryczny) i Michał „Redford" Kowalczyk (recenzent merytoryczny).

Ponadto autorzy pragną podziękować następującym osobom: Erik i Alessia, Ange Albertini, Bartłomiej Balcerek, Dawid Bałut, Richard Baranyi, Sergey Bratus, Marta Bukowska, Tadeusz Bukowski, Silvio Cesare, Anton Cherepanov, Arashi Coldwind, Samlis Coldwind, Anna Fałat, Olivier „Fafner [_KeyZee_]" Ferrand, Piotr Florczyk, Marcin Gabryszewski, Krzysztof Guc, Steven Hunter, Paweł Iwan, Przemek Jaroszewski, Nicolas Joly, Ewa Jurczyk, Jerzy Jurczyk, Paweł Kałuża, Krzysztof Katowicz-Kowalewski, Piotr Kijewski, Krzysztof Kołodziejski, Peter Košinár, Piotr Kramarczyk, Adam „Edisun" Kujawa, Tomasz Kuśmierski, Łukasz Kwiatek, Stanisław Litawa, Michał Ładanowski, Damian Malinowski, Osanda Malith, Marcin Noga, Grażyna Renkiewicz, Jérôme Segura, Tadeusz Składnikiewicz, Elżbieta Składnikiewicz, Tomasz Smolarek, Michał Spadliński, Piotr Szczepański, Kacper Szurek, Gavin Thomas, Julien Vanegue, Damian Wydro, Michał „lcamtuf" Zalewski, Marek Zmysłowski, a także Malware Hunter Team, Google Security Team, Dragon Sector, Vexillium oraz #szpica.

Rozdział 1

Funkcje, struktury, klasy i obiekty na niskim poziomie

Tomasz Bukowski

1.1. Wywoływanie funkcji w językach (bardzo) niskiego poziomu

Zdefiniujmy na początek kilka podstawowych pojęć odnoszących się do architektur x86 i x86-64, którymi będziemy się posługiwać w tym rozdziale:

- wywołanie funkcji – przekazanie kontroli do innego miejsca w kodzie z jednoczesnym odłożeniem na stos adresu następnej instrukcji;
- powrót z funkcji – pobranie ze stosu wartości IP oraz wykonanie skoku pod ten adres;
- *caller* – wywołujący – określa miejsce wywołania danej funkcji;
- *callee* – wywoływany – określa funkcję, która jest wywoływana.

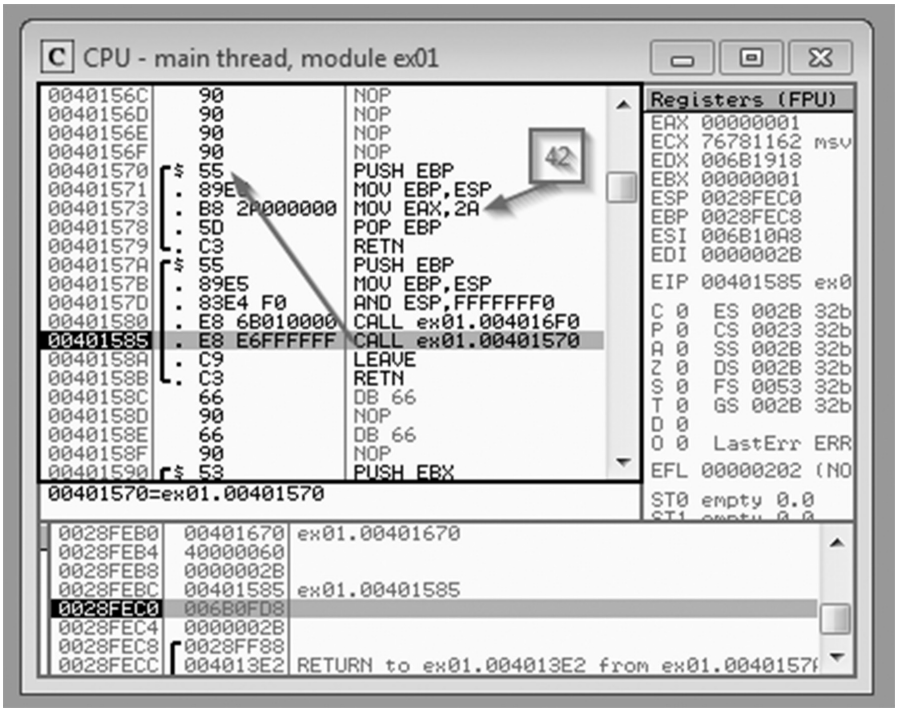
1.1.1. CALL, RET i konwencje wywołań

Podczas analizy kodu maszynowego (przetłumaczonego na asembler) trzeba zapomnieć o „dobrodziejstwach", jakie niosą za sobą języki wyższego poziomu, zaczynając już od pewnych podstawowych pojęć, takich jak funkcje. Dużo łatwiej zrozumieć działanie procesora, jeżeli kod (który jest mu dostarczony do wykonania) potraktujemy jako bliżej niepogrupowany szereg instrukcji. W tym miejscu należy wspomnieć o tym, że prawie każda architektura dostarcza prymitywnych namiastek funkcjonalności „wywoływania funkcji". W każdym znanym mi przypadku ogranicza się ona do instrukcji `CALL` lub równoważnej, której działanie można podzielić na dwa etapy: zapisanie (w pewnym miejscu) adresu następnej instrukcji, a następnie skok pod adres podany jako parametr instrukcji. W przypadku x86/64 „miejscem" tym jest stos. Instrukcją komplementarną do `CALL` jest instrukcja `RET`. Zdejmuje ona ze stosu wartość rejestru IP, odłożoną tam wcześniej podczas wywoływania funkcji. Dodatkowo, jeżeli dla instrukcji `RET` został podany parametr N, wskaźnik stosu SP zostanie zwiększony o tę wartość po zdjęciu adresu powrotu (co jest równoznaczne ze zdjęciem N bajtów ze stosu).

W listingu 1.1 przedstawiony został kod źródłowy testowego programu „ex01". Przełącznik `-O0` jest używany przy kompilacji, aby kompilator nie optymalizował kodu. Przełącznik `-m32` wymusza z kolei wygenerowanie kodu 32-bitowego.

```
// Kompilacja: gcc ex01.c -o ex01.exe -O0 -m32

int func1(void){
  return 42;
}
int main(void){
  return func1();
}
```

Listing 1.1. Kod programu „ex01”



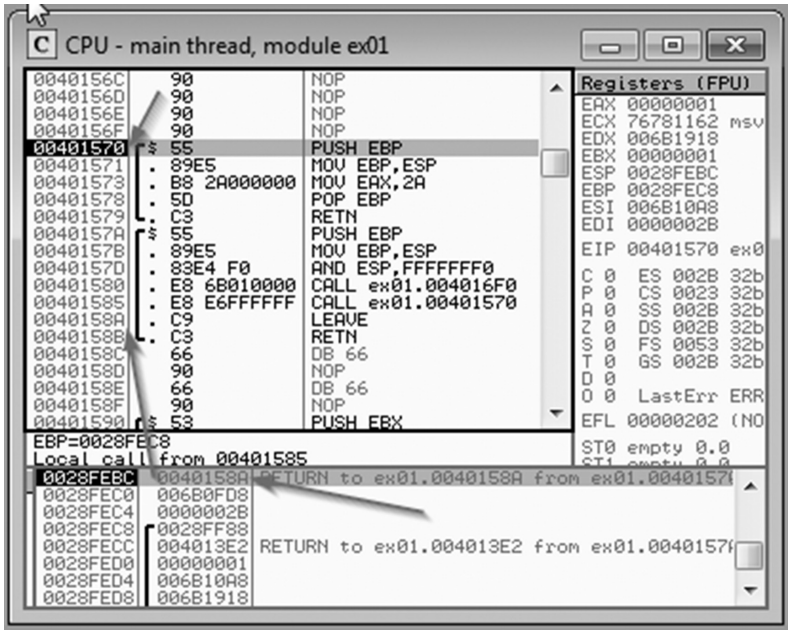
Rysunek 1.1. Parametr wskazuje nowe miejsce w kodzie przed wywołaniem instrukcji `CALL`

Nasuwa się pytanie, jak zatem przy użyciu tak „ubogiego” mechanizmu można wywołać funkcję w „nowoczesny” sposób, np:

```
x = moja_funkcja(1, 2, "string", &wskaznik_gdziekolwiek);
```

czyli przekazać wiele parametrów różnego typu oraz odczytać zwróconą przez nią wartość?

Można to osiągnąć na wiele sposobów, korzystając z dostępnych narzędzi, czyli rejestrów procesora i pamięci (w szczególności stosu). Problem zaczyna się, kiedy z naszego (binarnego) kodu musi skorzystać ktoś inny. Wtedy należy mu dostarczyć pewien opis interfejsu, tzw. ABI (*Application Binary Interface*), zawierający informację, w jaki sposób



Rysunek 1.2. Po wywołaniu adres powrotu jest odłożony na stosie

```
.text:00401570 ; =============== S U B R O U T I N E =======================================
.text:00401570
.text:00401570 ; Attributes: bp-based frame
.text:00401570
.text:00401570                 public _func1
.text:00401570 _func1          proc near               ; CODE XREF: _main+B↓p
.text:00401570                 push    ebp
.text:00401571                 mov     ebp, esp
.text:00401573                 mov     eax, 2Ah
.text:00401578                 pop     ebp
.text:00401579                 retn
.text:00401579 _func1          endp
.text:00401579
.text:0040157A
.text:0040157A ; =============== S U B R O U T I N E =======================================
.text:0040157A
.text:0040157A ; Attributes: bp-based frame
.text:0040157A
.text:0040157A ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:0040157A                 public _main
.text:0040157A _main           proc near               ; CODE XREF: ___tmainCRTStartup+25D↑p
.text:0040157A
.text:0040157A argc            = dword ptr  8
.text:0040157A argv            = dword ptr  0Ch
.text:0040157A envp            = dword ptr  10h
.text:0040157A
.text:0040157A                 push    ebp
.text:0040157B                 mov     ebp, esp
.text:0040157D                 and     esp, 0FFFFFFF0h
.text:00401580                 call    ___main
.text:00401585                 call    _func1
.text:0040158A                 leave
.text:0040158B                 retn
.text:0040158B _main           endp
.text:0040158B
.text:0040158B ; -------------------------------------------------------------------------
```

Rysunek 1.3. Wywołanie funkcji za pomocą instrukcji `CALL`, widok programu IDA Pro

autor kodu oczekuje wywoływania jego funkcji – czyli konwencję wywołań (ang. *calling convention*). W szczególności taka konwencja zawiera:

1. Listę i sposób przekazywania parametrów. W tym celu zazwyczaj wykorzystywane są określone rejestry oraz stos.
2. Sposób zwracania wartości (wyniku działania funkcji).
3. Listę rejestrów procesora, które nie zostaną zmienione po wykonaniu funkcji (tzw. bezpieczne rejestry). Najczęściej tożsame jest to z zapisaniem ich na stosie w prologu funkcji, a następnie zdjęciem w epilogu.
4. Ustalenie, kto „sprząta" (na stosie) po zakończeniu funkcji:
   – funkcja wywoływana: najczęściej przez wywołanie instrukcji `RET` z właściwym parametrem, co powoduje odpowiednie zwiększenie wartości wskaźnika stosu;
   – kod wywołujący funkcję: po instrukcji `CALL` następuje *explicite* „poprawienie" wartości wskaźnika stosu o odpowiednią wartość.

Jak można się domyślić, na większości platform istnieją już dobrze zdefiniowane konwencje, rozwiązujące ten problem za programistów, którzy dzięki temu mogą skupić się na właściwej logice kodu. W poniższych punktach omówiono szczegółowo konwencje obowiązujące w popularnych systemach działających pod kontrolą procesorów z rodziny x86 i x86-64.

1.1.2. Konwencje wywołań x86

Ponieważ platforma x86 rozwijała się bardzo burzliwie – od wczesnych lat 90. powstało wiele systemów operacyjnych, a jeszcze więcej kompilatorów – nic więc dziwnego, że nie ma jednej idealnej (i preferowanej przez wszystkich) konwencji wywołań. W tabeli 1.1 przedstawiono pięć najbardziej popularnych z nich. Kolumny zawierają kolejno opis:

1. Przyjętej nazwy potocznej.
2. Sposobu przekazywania parametrów poprzez rejestry oraz stos, w szczególności:
   a. W konwencji języka C – *C-style* („od tyłu") – argumenty odkładane są na stos od ostatniego do pierwszego:

      ```
      f(a1, a2) -> push a2, push a1, call
      ```

   b. W konwencji Pascala – *Pascal-style* („od przodu") – argumenty odkładane są na stos od pierwszego do ostatniego.
3. Kto sprząta stos: sama funkcja, czy też kod wywołujący.

We wszystkich przedstawionych konwencjach uznaje się, że funkcja wynik swojego działania zapisuje na koniec w rejestrze EAX (lub ST0 w przypadku wartości zmiennoprzecinkowej). Przyjmuje się również, że funkcja podczas działania może w pełni korzystać z wszystkich rejestrów, z tym zastrzeżeniem, że EBX, ESI, EDI oraz EBP po zakończeniu funkcji muszą mieć przywróconą oryginalną wartość (wartość przed rozpoczęciem wykonania kodu funkcji).

Tabela 1.1. Spis najbardziej popularnych konwencji wywołań na platformie x86

Nazwa	Parametry w rejestrach	Parametry na stosie	Kto sprząta stos	Opis
cdecl	brak	C-style	*caller*	Nazwa pochodzi od ***C** style function **decl**aration*. Pozwala na zmienną liczbę argumentów (tzw. *variadic functions*), ponieważ to wywołujący kod sprząta stos
pascal	brak	Pascal-style	*callee*	
stdcall	brak	C-style	*callee*	
fastcall	ECX, EDX	C-style	*callee*	Pierwsze dwa parametry są przekazywane w rejestrach, a pozostałe (jeśli istnieją) na stosie
thiscall	ECX	C-style	*callee*	Używana do wywoływania metod obiektów. W ECX znajduje się wskaźnik na obiekt, którego metodę wywołano

```
// Kompilacja: gcc ex02.c -o ex02.exe -O0 -m32

int __cdecl    func1(int a, int b, int c){ return a+b+c; }
int __stdcall  func2(int a, int b, int c){ return a+b+c; }
int __fastcall func3(int a, int b, int c){ return a+b+c; }
int main(void){
  int r=0;
  r += func1(1, 2, 3);
  r += func2(4, 5, 6);
  r += func3(7, 8, 9);
  return r;
}
```

Listing 1.2. Kod programu „ex02" przedstawiającego trzy różne konwencje wywołań

```
.text:00401570     push    ebp
.text:00401571     mov     ebp, esp
.text:00401573     mov     edx, [ebp+arg_0]
.text:00401576     mov     eax, [ebp+arg_4]
.text:00401579     add     edx, eax
.text:0040157B     mov     eax, [ebp+arg_8]
.text:0040157E     add     eax, edx
.text:00401580     pop     ebp
.text:00401581     retn
```

Rysunek 1.4. Deasemblacja kodu funkcji w konwencji *cdecl*

```
.text:00401582     push    ebp
.text:00401583     mov     ebp, esp
.text:00401585     mov     edx, [ebp+arg_0]
.text:00401588     mov     eax, [ebp+arg_4]
.text:0040158B     add     edx, eax
.text:0040158D     mov     eax, [ebp+arg_8]
.text:00401590     add     eax, edx
.text:00401592     pop     ebp
.text:00401593     retn    0Ch
```

Rysunek 1.5. Deasemblacja kodu funkcji w konwencji *stdcall*; widoczne „sprzątanie" stosu w epilogu

```
.text:00401596     push    ebp
.text:00401597     mov     ebp, esp
.text:00401599     sub     esp, 8
.text:0040159C     mov     [ebp+var_4], ecx
.text:0040159F     mov     [ebp+var_8], edx
.text:004015A2     mov     edx, [ebp+var_4]
.text:004015A5     mov     eax, [ebp+var_8]
.text:004015A8     add     edx, eax
.text:004015AA     mov     eax, [ebp+arg_0]
.text:004015AD     add     eax, edx
.text:004015AF     leave
.text:004015B0     retn    4
```

Rysunek 1.6. Deasemblacja kodu funkcji w konwencji *fastcall*; strzałkami zaznaczono dwa parametry przekazane przez rejestry

```
.text:004015C9     mov     [ebp+int_r], 0
.text:004015D0     mov     [esp+24h+var_1C], 3
.text:004015D8     mov     [esp+24h+var_20], 2
.text:004015E0     mov     [esp+24h+var_24], 1
.text:004015E7     call     func1
.text:004015EC     add     [ebp+int_r], eax
.text:004015EF     mov     [esp+24h+var_1C], 6
.text:004015F7     mov     [esp+24h+var_20], 5
.text:004015FF     mov     [esp+24h+var_24], 4
.text:00401606     call    _func2@12
.text:0040160B     sub     esp, 0Ch
.text:0040160E     add     [ebp+int_r], eax
.text:00401611     mov     [esp+24h+var_24], 9
.text:00401618     mov     edx, 8
.text:0040161D     mov     ecx, 7
.text:00401622     call    @func3@12
.text:00401627     sub     esp, 4
.text:0040162A     add     [ebp+int_r], eax
```

Rysunek 1.7. Kod „ex02" skompilowany na platformę x86; widoczne różne konwencje wywołań (1 – *cdecl*, 2 – *stdcall*, 3 – *fastcall*)

1.1.3. Konwencje wywołań x86-64

Konwencje wywołań obowiązujące na 64-bitowej architekturze x86-64 przedstawia tabela 1.2.

Tabela 1.2. Spis konwencji wywołań na platformie x86-64

System	Parametry w rejestrach	Parametry na stosie	Kto sprząta stos	Rejestry bezpieczne
Windows	RCX, RDX, R8, R9	C-style	*caller*	RBX, RSI, RDI, RBP, R12-R15
Linux, BSD, OS X	RDI, RSI, RDX, RCX, R8, R9	C-style	*caller*	RBX, RBP, R12-R15

W celach testowych możemy ponownie skompilować program „ex02” na rozważaną architekturę:

```
> gcc ex02.c -o ex02_64.exe -O0 -m64
> file *exe
ex02.exe:    PE32 executable (console) Intel 80386, for MS Windows
ex02_64.exe: PE32+ executable (console) x86-64, for MS Windows
```

W tym przypadku, dla wszystkich trzech wywołań funkcji (niezależnie od zdefiniowanych dla nich atrybutów) został wygenerowany ten sam kod, co widać na rysunku 1.8.

```
.text:0000000000401614    mov     [rbp+int_r], 0
.text:000000000040161B    mov     r8d, 3
.text:0000000000401621    mov     edx, 2
.text:0000000000401626    mov     ecx, 1
.text:000000000040162B    call    func1
.text:0000000000401630    add     [rbp+int_r], eax
.text:0000000000401633    mov     r8d, 6
.text:0000000000401639    mov     edx, 5
.text:000000000040163E    mov     ecx, 4
.text:0000000000401643    call    func2
.text:0000000000401648    add     [rbp+int_r], eax
.text:000000000040164B    mov     r8d, 9
.text:0000000000401651    mov     edx, 8
.text:0000000000401656    mov     ecx, 7
.text:000000000040165B    call    func3
.text:0000000000401660    add     [rbp+int_r], eax
.text:0000000000401663    mov     eax, [rbp+int_r]
```

Rysunek 1.8. 64-bitowy kod programu „ex02”, w którym wywołania mają tę samą formę

1.2. Struktury

Jak widać, w języku niskiego poziomu implementacja tak prostego i podstawowego zagadnienia jak wieloargumentowe funkcje niekoniecznie jest prosta. Przejdźmy teraz do bardziej złożonego problemu, a mianowicie obsługi struktur. Należy w tym miejscu powtórzyć: dla procesora cała dostępna pamięć wygląda „płasko". Nie rozumie on (czytaj: nie potrafi interpretować) bloków kodu jako funkcji – jak również organizowania grup komórek pamięci (bajtów) w złożone struktury.

```
struct st001{
  int first;
  char blob[200];
  int x;
  char y;
  unsigned long int z;
  int last;
};
```

Listing 1.3. Przykładowa definicja struktury w języku C zawierająca elementy o różnych typach

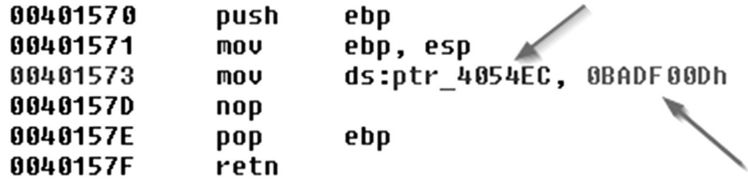
Pamięć potrzebna do przechowywania struktury może być zarezerwowana na dwa sposoby:

1. poprzez deklarację globalnej lub lokalnej zmiennej danego typu:
   a. zmienna lokalna – pamięć jest automatycznie rezerwowana na stosie podczas rozpoczęcia bloku funkcji,
   b. zmienna globalna – pamięć rezerwowana jest podczas inicjalizacji programu;
2. poprzez bezpośrednią alokację miejsca na strukturę (na przykład funkcją `malloc`).

Listingi 1.4, 1.5 i 1.6 oraz odpowiadające im rysunki 1.9, 1.10 i 1.11 przedstawiają zależność pomiędzy lokalizacją struktury w pamięci (w pamięci globalnej, na stosie i stercie) a odwołującym się do nich kodem asemblera.

```
#include "stru.h"
struct st001 global;
void func1(void){
  global.x = 0xbadf00d;
}
int main(void){
  func1();
  return 0;
}
```

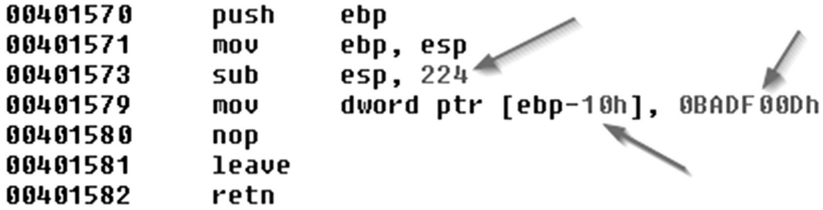
Listing 1.4. Przykładowy kod w języku C odwołujący się do pola struktury umieszczonej w pamięci statycznej



Rysunek 1.9. Wpisanie stałej pod odpowiedni adres w pamięci statycznej

```
#include "stru.h"
void func1(void){
  struct st001 local;
  local.x = 0xbadf00d;
}
int main(void){
  func1();
  return 0;
}
```

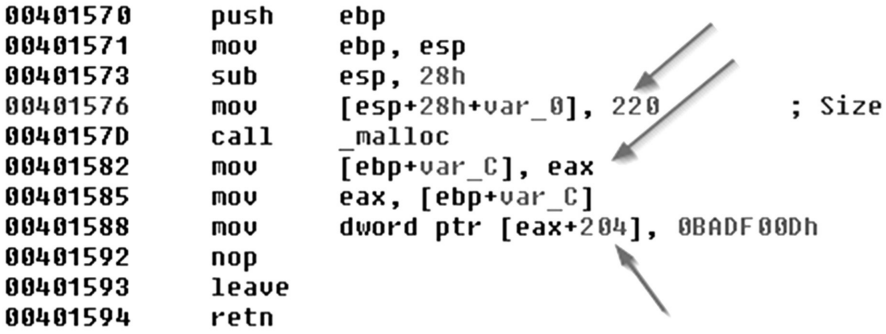
Listing 1.5. Przykładowy kod w języku C odwołujący się do pola struktury umieszczonej w lokalnej ramce stosu



Rysunek 1.10. Alokacja struktury w ramce stosu i wpisanie stałej pod odpowiednie przesunięcie (ang. *offset*)

```
#include "stru.h"
#include <stdlib.h>
void func1(void){
  struct st001* ptr_stru;
  ptr_stru = (struct st001*)malloc(sizeof(struct st001));
  ptr_stru->x = 0xbadf00d;
}
int main(void){ func1(); return 0; }
```

Listing 1.6. Przykładowy kod w języku C alokujący strukturę na stercie i odwołujący się do jej elementu



Rysunek 1.11. Alokacja struktury na stercie za pomocą funkcji `malloc`, a następnie odwołanie do odpowiedniego przesunięcia w otrzymanym wskaźniku

1.2.1. „Zgadywanie" wielkości i ułożenia elementów struktury w pamięci

Jak można zauważyć w powyższych przykładach, w przypadku programu, w którym wykorzystano struktury, czekają na nas dwa wyzwania:

1. Identyfikacja wielkości struktury – sytuacja jest stosunkowo prosta, jeżeli pamięć, w której przechowywane są dane, jest dynamicznie alokowana. Nieco trudniej jest, jeżeli mamy do czynienia z lokalną lub globalną zmienną.
2. Poznanie układu struktury – czyli identyfikacja lokalizacji poszczególnych pól. Zadanie to ułatwia nieco występowanie wielu zmiennych zajmujących pamięć „w okolicy" naszej struktury.

1.2.2. Rozpoznawanie budowy struktur lokalnych i globalnych

W listingu 1.7 znajduje się przykładowy kod ilustrujący wymienione powyżej problemy oraz etapy rozpoznania.

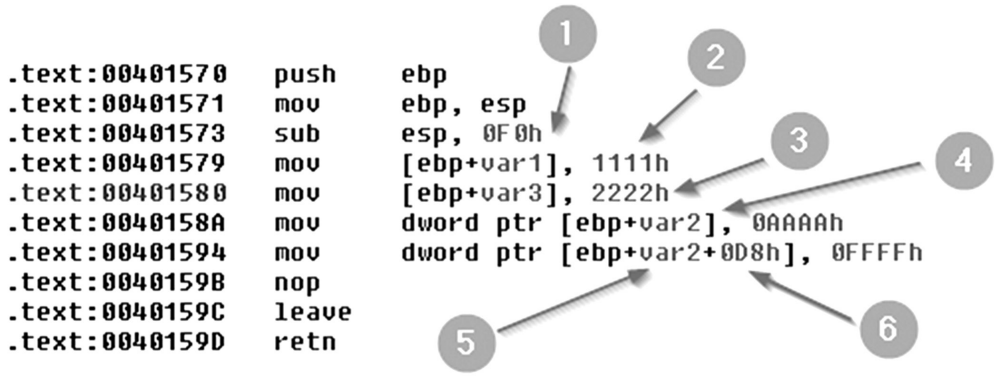
```
#include "stru.h"
void func1(void){
  int before;         //  rozmiar = 4B
  struct st001 local; //  nieznany rozmiar
  int after[4];       //  rozmiar = 16B
  before = 0x1111;
  after[0] = 0x2222;
  local.first = 0xaaaa;
  local.last = 0xffff;
}
```

```
int main(void){
  func1();
  return 0;
}
```

Listing 1.7. Kod w języku C operujący na lokalnej strukturze

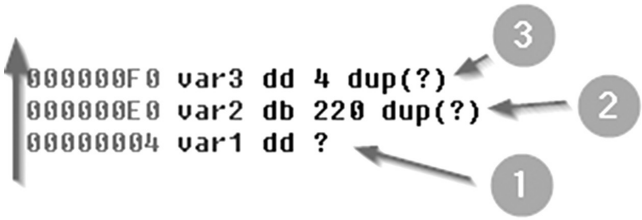
Załóżmy, że wiemy, iż lokalnie występują dwie zmienne o znanym rozmiarze oraz struktura „pomiędzy" nimi (w rozumieniu układu pamięci). Pierwsza zmienna ma wielkość czterech bajtów (`sizeof(int)`), a druga 16 (4 × `sizeof(int)`). Kiedy przeanalizujemy kod asemblera wygenerowany na podstawie przytoczonych źródeł w języku C (patrz rys. 1.12), możemy dostrzec następujące etapy wykonania funkcji `func1`:

1. Alokacja miejsca na stosie na zmienne lokalne, gdzie 0xF0 = 240 bajtów. Stąd można oszacować wielkość pamięci zarezerwowanej na strukturę: 240 − (4 + 16) = 220 bajtów.
2. Przypisanie stałej do zmiennej `before`.
3. Przypisanie stałej do elementu `after[0]`.
4. Przypisanie stałej do pierwszego pola struktury. Bardziej czytelny byłby tutaj zapis `[ebp+var2+0x0]`. Przykład ten pokazuje, że z punktu widzenia kodu niskopoziomowego pojęcie struktury (jako organizacji pamięci) nie istnieje.
5. Przypisanie stałej do ostatniego pola struktury, znajdującego się na przesunięciu 0xD8. Zapis ten jest widoczny tylko dlatego, że zmienną `var2` oznaczono (w programie do deasemblacji) jako tablicę 220 bajtów na stosie.



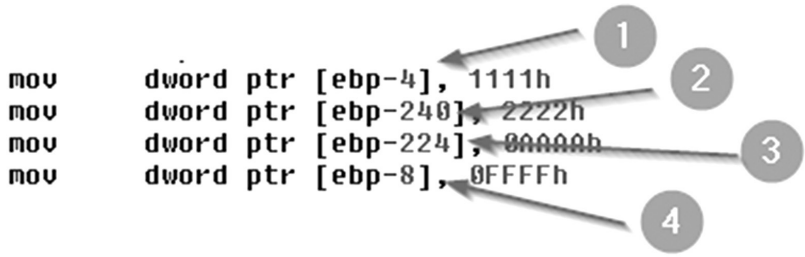
Rysunek 1.12. Kod asemblera funkcji `func1` przedstawionej w listingu 1.7

Układ pamięci na stosie funkcji pokazano na rysunku 1.13. Stos rośnie w dół, więc zmienne lokalne odłożone są w odwrotnej kolejności. Zmienne o znanym rozmiarze to `var1` (1) oraz `var3` (3). Element `var2` (kandydat na poszukiwaną strukturę) został oznaczony jako tablica o długości 220 bajtów (rozmiar „odgadnięty" na podstawie powyższej analizy miejsca zarezerwowanego na stosie).



Rysunek 1.13. Układ ramki stosu funkcji `func1`

Na rysunku 1.14 widzimy fragment kodu funkcji `func1` bez opisów ułatwiających analizę układu pamięci (zmiennych lokalnych na stosie). Odwołania do zmiennych lokalnych są podane względem adresu aktualnej ramki stosu (rejestr EBP). W tabeli 3 znajduje się dokładny układ zmiennych na stosie.



Rysunek 1.14. Przypisania stałych do komórek na stosie bez podanych opisów pomocniczych

Tabela 1.3. Dokładny układ obiektów na stosie

Nazwa	Przesunięcie	Opis
var3	240	Pierwszy element tablicy `var3`
	...	Kolejne elementy tablicy
var2	224	Pierwsze pole struktury
	...	Kolejne pola struktury
	8	Ostatnie pole struktury
var1	4	Zmienna `var1`

1.2.3. Rozpoznawanie budowy struktur dynamicznie alokowanych

W przypadku kodu, w którym występuje dynamiczna alokacja pamięci dla struktury i opcjonalnie inicjalizacja jej pól, sytuacja wygląda dużo prościej – z dwóch powodów:

1. Jak już opisano wcześniej – rozmiar struktury można poznać poprzez zlokalizowanie parametru przekazywanego funkcji alokującej pamięć (np. `malloc`).

2. Po wykonaniu alokacji program otrzymuje adres regionu pamięci, więc kod tak naprawdę operuje na wskaźniku do struktury. Operacje na poszczególnych polach widoczne są jako odwołania do wskaźnika z pewnym przesunięciem.

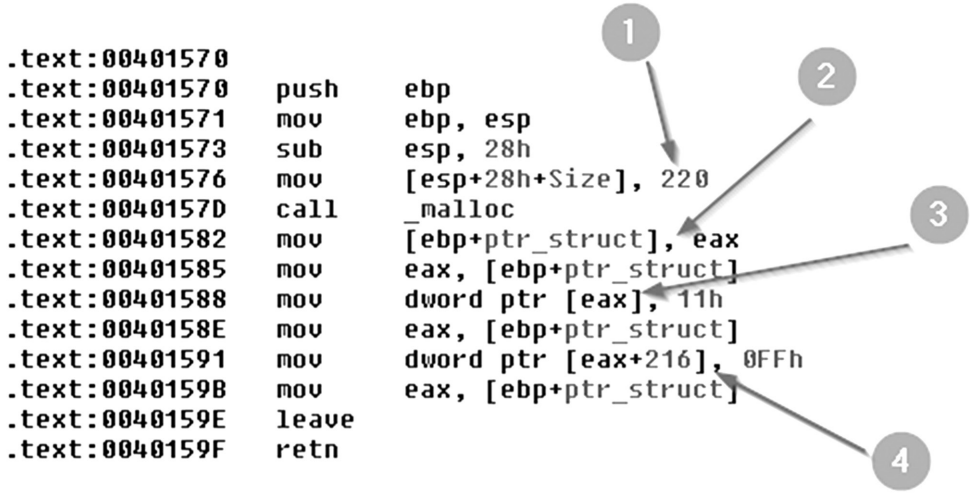
W listingu 1.8 przedstawiony jest kod z użyciem dedykowanej funkcji (`new_struct`), która alokuje pamięć na strukturę oraz ustawia wartości jej niektórych pól. Funkcja `func1` wykorzystuje otrzymany wskaźnik do przeprowadzania kolejnej operacji na strukturze.

```
#include <stdlib.h>
#include "stru.h"

struct st001* new_struct(void){
  struct st001* ps;
  ps = (struct st001*)malloc(sizeof(struct st001));
  ps->first = 0x11;
  ps->last = 0xFF;
  return ps;
}

void func1(void){
  struct st001* ptr;
  ptr = new_struct();
  ptr->x = 0x1234;
}

int main(void){
  func1();
  return 0;
}
```

Listing 1.8. Dynamiczna alokacja struktury i dostęp do jej elementów w języku C

Wynik deasemblacji funkcji `new_struct` jest widoczny na rysunku 1.15. Oto, co możemy wywnioskować z kodu asemblera:

1. Alokacja 220 bajtów pamięci ujawnia rozmiar struktury.
2. Wskaźnik do zaalokowanej pamięci przechowywany jest w zmiennej lokalnej.
3. Następuje odwołanie do pierwszego elementu struktury o rozmiarze czterech bajtów. Wskaźnik na strukturę przechowywany jest w rejestrze EAX. Bardziej czytelny zapis mógłby wyglądać tak: `[eax+0x0]`.
4. Następuje odwołanie do ostatniego elementu struktury (przesunięcie 216), również o szerokości czterech bajtów.



Rysunek 1.15. Wynikowy kod funkcji `new_struct`

1.3. Klasy, obiekty, dziedziczenie i tablice wirtualne

Skoro jesteśmy już w stanie rozpoznawać (do pewnego stopnia) struktury i ich układ w kodzie C/C++, spróbujmy podjąć się jeszcze trudniejszego zadania – analizy klas.

1.3.1. Prosta klasa a struktura

Kiedy początkujący programista – obyty już z koncepcją struktur – zapyta, co to są klasy, chyba najlepszą (obrazującą ideę, lecz bardzo upraszczającą) odpowiedzią byłoby stwierdzenie: „to takie struktury z funkcjami". Jeżeli popatrzymy na tę kwestię z perspektywy kodu niskopoziomowego, to niewiele minęliśmy się z prawdą. O ile struktury i funkcje zostały opisane osobno wcześniej, to połączenie obu mechanizmów w jeden wymaga pewnego objaśnienia. Spójrzmy najpierw na różnicę w kodzie wysokopoziomowym oraz niskopoziomowym dwóch podobnych programów, przedstawionych w listingach 1.9 i 1.10. Jeden jest napisany z użyciem struktury, drugi – obiektowo (z użyciem klasy).

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

typedef struct s_car {
  char desc[200];
  int color;
  int wheels;
```

```
  int doors;
} s_car;

typedef s_car* p_car;

void car__print(p_car car){
  printf("[color:%d wheels:%d doors:%d]\n", car->color, car->wheels, car->doors);
}

void car__set_color(p_car car, int c){
  printf("SET COLOR\n");
  car->color = c;
}

p_car car__new(void){
  p_car c = (p_car)malloc(sizeof(s_car));
  c->color = 0; c->wheels=0; c->doors=0;
  printf("Creating new instance...\n");
  return c;
}

int main(int argc, char* argv[]){
  printf("START\n");
  p_car c = car__new();
  car__print(c);
  car__set_color(c, 100);
  car__print(c);
  printf("END\n");
  return 0;
}
```

Listing 1.9. Program w języku C opisujący samochód przy użyciu struktury

```
#include <cstdio>
#include <cstdlib>

class c_car {
 public:
  char desc[200];
  int color;
  int wheels;
  int doors;
  c_car();
  void print();
  void set_color(int);
```

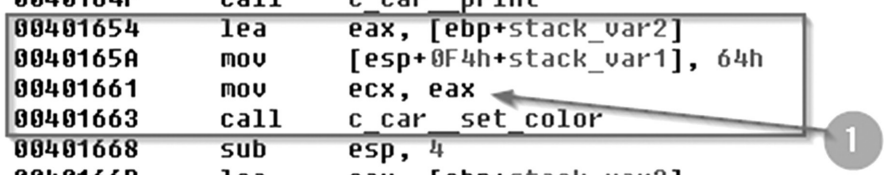
```
};

void c_car::print(){ // <-- METODA
  printf("[color:%d wheels:%d doors:%d]\n", color, wheels, doors);
}

void c_car::set_color(int c){
  printf("SET COLOR\n");
  color = c;
}

c_car::c_car(void){ // <-- KONSTRUKTOR
  printf("Creating new instance...\n");
  color=0; wheels=0; doors=0;
}

int main(int argc, char* argv[]){
  printf("START\n");
  c_car car;
  car.print();
  car.set_color(100);
  car.print();
  printf("END\n");
  return 0;
}
```

Listing 1.10. Program w języku C++ opisujący samochód przy użyciu klasy

```
00401634     call    car__new
00401639     mov     [esp+20h+stack_var2], eax
0040163D     mov     eax, [esp+20h+stack_var2]
00401641     mov     [esp+20h+stack_var1], eax
00401644     call    car__print
00401649     mov     [esp+20h+stack_var3], 64h
00401651     mov     eax, [esp+20h+stack_var2]
00401655     mov     [esp+20h+stack_var1], eax
00401658     call    car__set_color
0040165D     mov     eax, [esp+20h+stack_var2]
00401661     mov     [esp+20h+stack_var1], eax
00401664     call    car__print
```

Rysunek 1.16. Kod strukturalny – deasemblacja części głównej funkcji; wywołanie funkcji `set_color` w konwencji *stdcall*

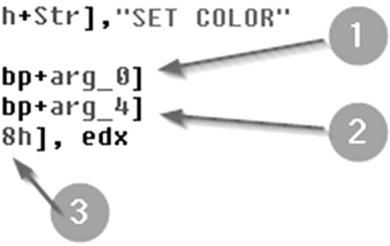
```
0040163A      lea     eax, [ebp+stack_var2]
00401640      mov     ecx, eax
00401642      call    c_car__constructor
00401647      lea     eax, [ebp+stack_var2]
0040164D      mov     ecx, eax
0040164F      call    c_car__print
00401654      lea     eax, [ebp+stack_var2]
0040165A      mov     [esp+0F4h+stack_var1], 64h
00401661      mov     ecx, eax
00401663      call    c_car__set_color
00401668      sub     esp, 4
0040166B      lea     eax, [ebp+stack_var2]
00401671      mov     ecx, eax
00401673      call    c_car__print
```



Rysunek 1.17. Kod obiektowy – deasemblacja części głównej funkcji; wywołanie funkcji `set_color` w konwencji *thiscall*

Sam kod funkcji ustawiającej kolor wygląda w obu przypadkach niemal tak samo (patrz rys. 1.18 i 1.19).

```
004015B2      mov     [esp+18h+Str],"SET COLOR"
004015B9      call    _puts
004015BE      mov     eax, [ebp+arg_0]
004015C1      mov     edx, [ebp+arg_4]
004015C4      mov     [eax+0C8h], edx
004015CA      nop
004015CB      leave
004015CC      retn
```



Rysunek 1.18. Kod strukturalny – deasemblacja części funkcji `set_color`; oba parametry funkcji (wskaźnik na strukturę oraz kolor) są przekazywane przez stos

```
004015B6      mov     [ebp+arg1], ecx
004015B9      mov     [esp+28h+Str],"SET COLOR"
004015C0      call    _puts
004015C5      mov     eax, [ebp+arg1]
004015C8      mov     edx, [ebp+arg2]
004015CB      mov     [eax+0C8h], edx
```

Rysunek 1.19. Kod obiektowy – deasemblacja części funkcji `set_color`; pierwszy argument (zgodnie z konwencją *thiscall*) znajduje się w rejestrze ECX

1.3.2. Obiekty = struktury + funkcje + thiscall

Jedyna różnica między dwiema wersjami powyższego programu polega (w zasadzie) na tym, że w przypadku obiektowym kompilator „wie", że trzeba (niejawnie dla programisty) przekazać wskaźnik obiektu jako pierwszy parametr (w konwencji *thiscall* – czyli w rejestrze ECX). W przypadku kodu strukturalnego programista musi zrobić to „ręcznie" – przekazując wskaźnik jako parametr. Odnalezienie kodu, w którym podczas wywoływania funkcji użyto konwencji *thiscall*, jest pierwszą i w zasadzie najważniejszą wskazówką świadczącą o tym, że prawdopodobnie mamy do czynienia z kodem obiektowym – a funkcje wywoływane w ten sposób są metodami pewnej klasy.

Naturalnym wydawać by się mogło, że metody klasy będą na etapie kompilacji zapisywane tak samo jak elementy struktury – jako wskaźniki na funkcje. Jak się jednak okazuje – nie ma takiej potrzeby. Przede wszystkim dlatego, że kompilator podczas przetwarzania kodu do postaci binarnej dobrze „wie", do jakiej klasy należy obiekt i jaką funkcję trzeba wywołać jako jego metodę (pamiętając o *this* w rejestrze ECX).

Nie zawsze jednak...

1.3.3. Wszystko zostaje w rodzinie, czyli dziedziczenie

Istnieje jeszcze jeden przypadek wymagający rozważenia – a mianowicie dziedziczenie oraz metody wirtualne. Problem dobrze ilustruje kod pokazany w listingu 1.11.

```
#include <cstdio>

class Base {
  int prop;
 public:
  virtual void func1(void){ puts("-> F1 (base)"); }
  virtual void func2(void){ puts("-> F2 (base)"); }
  Base(){  puts("NEW: Base"); }
  virtual ~Base(){ puts("DEL: Base"); }
};

class Child1 : public Base {
 public:
  Child1(){ puts("NEW: Child1"); }
  virtual void func1(void){ puts("-> F1 (child)"); }
};

class Child2 : public Base {
 public:
  virtual void func2(void){ puts("-> F2 (child)"); }
};
```

```
void do_stuff(Base *o){
  o->func1();
  o->func2();
  delete o;
}

void work(void){
  puts(" -- object 1 --");
  do_stuff(new Child1());
  puts(" -- object 2 --");
  do_stuff(new Child2());
}

int main(void){
  work();
  return 0;
}
```

Listing 1.11. Kod programu demonstrującego dziedziczenie oraz metody wirtualne

Klasa główna `Base` ma dwie metody: `func1` oraz `func2`. Każda z klas potomnych `Child1` oraz `Child2` implementuje (przeciąża) jedną z nich.

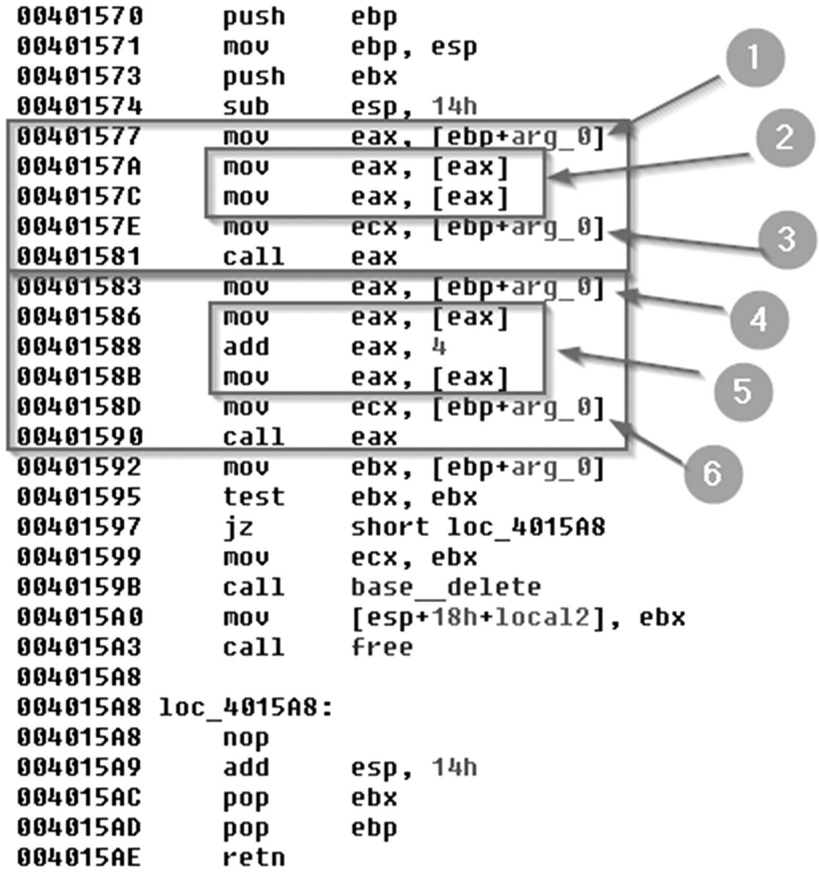
W tej sytuacji funkcja `do_stuff` przyjmuje wskaźnik na obiekt klasy „matki" (`Base`), a następnie wywołuje jego dwie metody i niszczy go. Jak można się przekonać – pomimo przekazania referencji do obiektu klasy `Base` – wykonywany jest kod przeciążonych funkcji. Kompilator na etapie budowania kodu binarnego nie wie, jaki obiekt zostanie przekazany jako argument funkcji `do_stuff` – a tym samym nie wie, które konkretnie z metod powinny zostać wywołane. Patrząc na listing 1.12, widzimy, iż program działa zgodnie z oczekiwaniami. Zatem w jakiś tajemniczy sposób przekazywana jest informacja (zapewne za pomocą wskaźnika – jak to bywa w kodzie maszynowym) o funkcji, która ma zostać wywołana.

```
> g++ inh.cpp -o inh -m32
> ./inh.exe
 -- object 1 --
NEW: Base
NEW: Child1
-> F1 (child)
-> F2 (base)
DEL: Base
 -- object 2 --
NEW: Base
-> F1 (base)
-> F2 (child)
DEL: Base
```

Listing 1.12. Kompilacja oraz uruchomienie programu prezentującego dziedziczenie

Przyjrzyjmy się bliżej deasemblacji funkcji przedstawionej na rysunku 1.20. Ramkami zaznaczono wywołania `func1` oraz `func2`. Tuż przed wywołaniem do rejestru ECX przypisywana jest wartość pierwszego (jedynego) parametru funkcji `do_stuff`. Widać również pewne operacje wykonywane przed samą instrukcją `CALL` – „tajemnicze” odnajdywanie wskaźnika do funkcji, którą należy wywołać. Analizując poszczególne etapy po kolei widzimy, że:

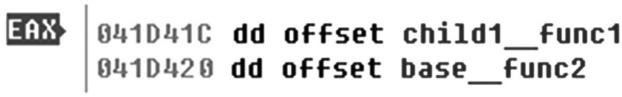
1. Do rejestru EAX przypisywana jest wartość `arg_0`.
2. Do rejestru EAX przypisywana jest wartość wskazywana przez rejestr EAX (dwukrotnie).
3. Adres, na który wskazuje rejestr EAX, zostaje wywołany jako funkcja.



Rysunek 1.20. Wynik deasemblacji funkcji `do_stuff` w przykładzie z listingu 1.11

Powyższy zapis jest jednak trudny do zrozumienia. Spróbujmy przeanalizować, co znajduje się w pamięci (rys. 1.21 i 1.22).

Rysunek 1.21. Wartości zapisane w pamięci pod adresem wskazywanym przez rejestr EAX po wykonaniu instrukcji (1)



Rysunek 1.22. Wartości zapisane w pamięci pod adresem wskazywanym przez rejestr EAX po wykonaniu instrukcji (2)

Widzimy tutaj, że po wykonaniu wszystkich odwołań do pamięci mamy ostatecznie do czynienia z tablicą wskaźników na funkcje. Jest to tablica funkcji wirtualnych (zwana częściej *vtable* lub VMT). Wskaźnik do tej tablicy umieszczany jest w dodatkowym polu w pamięci klasy, które znajduje się z reguły na jej samym początku (pod przesunięciem 0). Wiedząc o tym, możemy raz jeszcze spróbować opisać zdeasemblowany kod funkcji `do_stuff`:

1. Do rejestru EAX przypisywany jest wskaźnik na obiekt (*this*).
2. Pozyskiwany jest wskaźnik na właściwą funkcję:
   a. Pobierana jest wartość pierwszego elementu struktury, czyli adres vtable.
   b. Pobierany jest wskaźnik z pierwszego elementu tablicy vtable, czyli adres funkcji `func1` (dla klasy `Child1`).
5. Do rejestru ECX przypisywany jest wskaźnik na obiekt (*this*), a następnie ma miejsce wywołanie metody (na którą wskazuje rejestr EAX).

Dla drugiego wywołania analogicznie:

4. Do rejestru EAX przypisywany jest wskaźnik na strukturę obiektu (*this*)
5. Pozyskiwany jest wskaźnik na właściwą funkcję:
   a. Pobierana jest wartość pierwszego elementu struktury, czyli adres vtable.
   b. Pobierany jest wskaźnik z drugiego elementu tablicy vtable, czyli adres funkcji `func2`.
6. Do rejestru ECX przypisywany jest wskaźnik na obiekt (*this*), po czym następuje wywołanie metody (na którą wskazuje rejestr EAX).

1.4. Podsumowanie

Wykonując analizę wsteczną skompilowanego kodu obiektowego, powinniśmy kierować się poniższymi wskazówkami:

1. Szukamy konstruktora. Pozwoli on oszacować wielkość struktury obiektu.

   a. Jeżeli do pierwszego pola struktury w konstruktorze została przypisana wartość wskazująca na tablicę wskaźników funkcji – mamy do czynienia z vtable, a nasz obiekt zawiera metody wirtualne.
2. Szukamy i analizujemy metody funkcji, patrząc, gdzie przekazywany jest wskaźnik na obiekt z wykorzystaniem rejestru ECX (w przypadku kodu 32-bitowego). Jeżeli obiekt zawiera vtable – funkcje te możemy znaleźć właśnie w tej tablicy. Analiza metod obiektu pozwoli na lepsze dopasowanie konkretnych pól klasy.
3. Jeżeli wskaźnik do pewnej funkcji znajduje się w więcej niż jednej tabeli metod wirtualnych – mamy do czynienia z dziedziczeniem.

Rozdział 2

Środowisko uruchomieniowe na systemach GNU/Linux

Grzegorz Antoniak

2.1. Wstęp

Systemy operacyjne oparte na jądrze Linuksa oferują wiele metod ułatwiających analizę oprogramowania pod kątem jego działania. Większość popularnych mechanizmów wykorzystywanych z powodzeniem na systemach Windows dostępnych jest też i tutaj, choć często w nieco innej formie.

W tym rozdziale będziemy stosować system nazewnictwa narzędzi typu `nazwa(N)`, gdzie N to numer lub identyfikator. Po napotkaniu takiej nazwy, można szybko przejść do dokumentacji tego narzędzia przy użyciu programu *man*, poprzez wykonanie polecenia:

```
$ man N nazwa
```

Na przykład aby uzyskać informacje na temat narzędzia `readelf(1)`, należy wykonać polecenie:

```
$ man 1 readelf
```

Taka lista argumentów powoduje otwarcie podręcznika dotyczącego narzędzia `readelf` z sekcji pierwszej, czyli programów narzędziowych.

2.2. Pliki wykonywalne ELF

Programy wykonywalne oraz biblioteki współdzielone (ang. *shared libraries* lub *shared objects*) w systemach GNU/Linux zapisywane są w plikach, których struktura zdefiniowana jest przez standard ELF (*Executable and Linkable Format*).

GNU/Linux nie są jedynymi systemami, w których wykorzystano format ELF do przechowywania treści plików wykonywalnych lub bibliotek; jest on stosowany również w:

- innych systemach uniksowych (rodzina systemów BSD: FreeBSD, OpenBSD, NetBSD, Solaris, QNX itp.),
- alternatywnych systemach użytkowych lub badawczych (AmigaOS 4, MorphOS, AROS, Plan9),
- systemach konsoli gier komputerowych (Sony Playstation 2/3/4, Nintendo Wii),

- urządzeniach podręcznych, takich jak tablety, smartfony z systemami Android, Samsung Bada, Nokia Maemo itp.

Użycie tego formatu na wielu różnych systemach wynika z faktu, że ELF został zaprojektowany jako format uniwersalny. To – niestety – czasem wiąże się z tym, że niektóre wartości zapisane w pliku będą miały sens tylko wtedy, gdy będą interpretowane zgodnie z regułami interpretacji określonymi dla danej architektury. Na szczęście takich specyficznych miejsc jest niewiele i dużą część formatu można zinterpretować za pomocą ogólnych metod.

Uniwersalność formatu ELF pociąga za sobą konieczność identyfikacji typu danych przechowywanych w pliku. W zależności od typu architektury, dla której przeznaczony jest program, format części danych (takich jak kod źródłowy) będzie nieco inny. Na przykład program na konsolę PlayStation 3 przechowywany w pliku ELF będzie zawierał inny kod maszynowy niż program w pliku ELF przeznaczony dla architektury x86-64; przyczyna jest prosta – obie architektury różnią się od siebie i zawierają różne jednostki centralne (procesory).

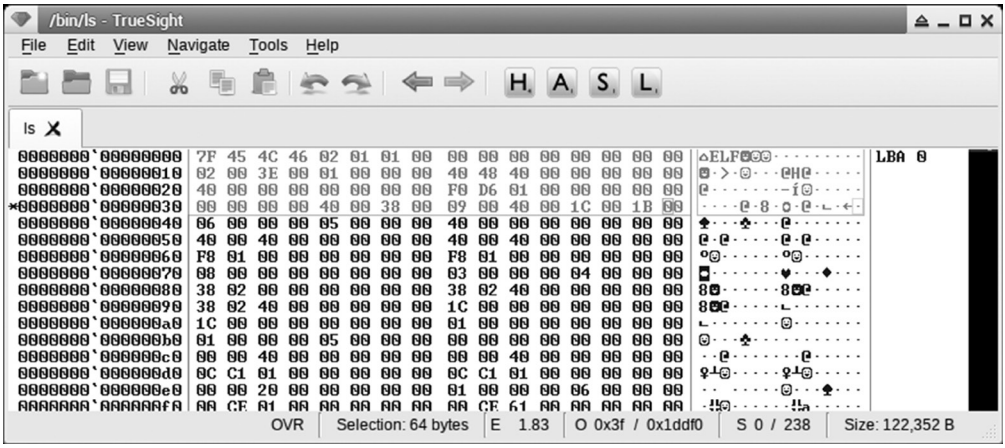
Do analizy struktury pliku ELF najczęściej wykorzystuje się narzędzia `readelf(1)` i `objdump(1)`, dostępne właściwie w każdej dystrybucji systemu GNU/Linux (wchodzą w skład popularnego pakietu narzędziowego binutils). W dalszych częściach rozdziału będziemy się posługiwać tymi programami do odczytywania strategicznych informacji z przykładowych plików wykonywalnych lub bibliotek.

Pełne definicje formatu ELF znajdują się w pliku /usr/include/linux/elf.h, który jest dostępny w większości dystrybucji (może być wymagane doinstalowanie odpowiedniego pakietu – w dystrybucji ArchLinux pakiet ten nosi nazwę linux-api-headers) oraz w dokumentacji [3]. Polecamy zapoznanie się z treścią tego pliku, ponieważ rozdział ten nie powstał z myślą o przedstawieniu kompletnej dokumentacji tego standardu; pokazane w nim będą jedynie najważniejsze aspekty formatu, przydatne podczas przeprowadzania analizy plików wykonywalnych o zamkniętym źródle.

2.2.1. Identyfikacja systemu i architektury docelowej

Jedną z najbardziej podstawowych informacji, którą należy określić zaraz na samym początku, jest architektura docelowa programu. Jeśli kod jest kierowany pod procesor PowerPC, nie ma sensu uruchamiać go na architekturze x86-64 i odwrotnie. Często analizując sam plik binarny, można określić dokładną architekturę platformy docelowej, dla której dany plik ELF był zbudowany. Posiadanie tej informacji umożliwi poprawną interpretację kodu maszynowego zapisanego w odpowiednich sekcjach.

Poszukiwane dane znajdują się w nagłówku o nazwie *file header* (struktury `Elf32_Ehdr` lub `Elf64_Ehdr`), który znajduje się zaraz na początku pliku. Zrzut ekranu na rysunku 2.1 przedstawia nagłówek *file header* w programie do edycji danych binarnych.



Rysunek 2.1. Nagłówek *file header*

Adres 0x00 zawiera 4 bajty, które zawsze przyjmują tę samą postać: 0x7F, 0x45, 0x4C oraz 0x46. W przypadku, gdy plik pod tym adresem zawiera inną treść, najprawdopodobniej nie jest to plik ELF.

- Adres 0x04 zawiera jeden bajt identyfikujący długość słowa platformy docelowej. Bajt o wartości „1" identyfikuje platformę 32-bitową, natomiast bajt o wartości „2" oznacza platformę 64-bitową. Należy zwrócić uwagę, że w przypadku platformy 64-bitowej wszystkie adresy będą zajmowały 8 bajtów, zamiast standardowych 4 bajtów na platformie 32-bitowej. Powoduje to często sytuację, w której adresy poszczególnych pól mogą się różnić pomiędzy wersją 32- a 64-bitową.
- Adres 0x05 zawiera jeden bajt określający kolejność bajtów na danej platformie. Są dostępne dwie możliwości: najbardziej znaczący bajt jako pierwszy (ang. *big-endian*, np. architektura PowerPC) symbolizowaną przez wartość „1" oraz najbardziej znaczący bajt jako ostatni (ang. *little-endian*, np. architektura x86) symbolizowaną przez wartość „2".
- Adres 0x07 zawiera jeden bajt określający użyty interfejs binarny (*Application Binary Interface*, w skrócie *ABI*). Nowe wersje specyfikacji ELF wycofują użycie tego pola i obecnie zaleca się jego wyzerowanie. Warto jednak zwrócić uwagę na informacje znajdujące się w tym polu; historycznie wiele systemów wpisywało tutaj dedykowaną wartość wskazującą na rodzaj systemu, pod który kierowany był plik wykonywalny:
  - wartość „0" zgodnie ze standardem oznaczała wartość niesprecyzowaną;
  - wartość „1" oznaczała system HP-UX (spadkobierca System V, produkowany przez firmę HP po 1984 r.); aktualnie wartość ta jest wykorzystywana w systemie GNU/Linux;
  - wartość „2" oznaczała system NetBSD;
  - wartość „3" była zarezerwowana dla systemu GNU/Linux, nie jest już jednak stosowana;

  - wartość „4” może oznaczać system GNU/Hurd, tworzony przez projekt GNU;
  - wartość „6” oznaczała system Solaris, produkowany przez firmę Oracle (wcześniej Sun Microsystems);
  - wartość „7” była stosowana do określenia ABI systemu AIX (jest to system UNIX produkowany przez firmę IBM);
  - wartość „9” jest nadal wykorzystywana w systemie FreeBSD;
  - wartość „12” była dedykowana dla systemu OpenBSD.
- Adres 0x08 historycznie zawierał informacje o wersji ABI zdefiniowanego w poprzednim polu. Aktualna specyfikacja formatu ELF zaleca wyzerowanie tego pola.
- Adres 0x10 zawiera dwa bajty określające typ pliku wykonywalnego. Wartość „0” to niesprecyzowany typ – wartość ta nigdy nie powinna pojawić się w pliku. Wartość „1” oznacza typ relokowany i jest stosowana do opisu plików obiektów powstałych jako wynik kompilacji np. kompilatora C lub C++ (pliki *.o). Wartość „2” (`ET_EXEC`) oznacza standardowy plik wykonywalny. Wartość „3” (`ET_DYN`) jest zarezerwowana dla bibliotek, ale również dla programów wykonywalnych. Wartość „4” oznacza specjalny plik core, zawierający informacje o stanie programu w momencie jego awarii.
  Przy interpretacji danych z tego pola warto zwrócić uwagę na to, że wartość „3” (`ET_DYN`) może być stosowana do oznaczenia zarówno plików wykonywalnych, jak i bibliotek, przy czym jego wykorzystanie przy plikach wykonywalnych zaczęło być bardziej popularne stosunkowo niedawno. Programy oznaczone w ten sposób muszą mieć kod PIC (*Position Independent Code*, któremu będzie poświęcony jeden punkt). Wykorzystanie `ET_DYN` do opisu programów wykonywalnych jest popularne m.in. na nowszych systemach OpenBSD, znany jest też przypadek generowania tak oznaczonych plików wykonywalnych przez kompilator języka Rust.
- Adres 0x12 przechowuje dwa bajty identyfikujące architekturę docelową. Może przyjmować wiele wartości, wśród których najpopularniejsze to: wartość „2” dla architektury SPARC, wartość „3” dla architektury x86 (32-bitowa), „8” dla architektury MIPS (stosowana m.in. w routerach), „40” dla architektury ARM (urządzenia wbudowane, tablety lub smartfony) oraz „62” dla architektury x86-64 (64-bitowa).

Szybką identyfikację architektury docelowej można uzyskać za pomocą narzędzia `file(1)`:

```
$ file /bin/ls
/bin/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=0a07d110567fac845adae753f7f8561a4121279d, stripped
```

W skład wyświetlonych przez `file(1)` informacji wchodzi większość danych, które pozwalają na dobór odpowiednich narzędzi do dalszej analizy:

- `ELF 64-bit LSB executable` – informacje o oznaczeniu `ET_EXEC` do opisu pliku wykonywalnego oraz informacje o długości słowa (64 bity) i kolejności bajtów (LSB – *least significant bit*, czyli kodowanie *little-endian*).

- `x86_64` – architektura x86-64.
- `dynamically linked` – w pliku istnieje sekcja `PT_DYNAMIC`, opisana w punkcie 2.2.5, „Sekcja .dynamic", która pozwala na zdefiniowanie bibliotek zależnych wykorzystywanych przez program. Alternatywnym odczytem jest `statically linked`, który informuje o braku tej sekcji.
- `interpreter [...]` – ścieżka do programu ładującego (w tym przypadku ld.so), którego zadaniem jest poprawne załadowanie pliku ELF do pamięci, wraz z jego wszystkimi zależnościami. Program ładujący ld.so zostanie opisany w późniejszym podrozdziale.
- `BuildID[sha1]=[...]` – unikatowy identyfikator pliku wykonywalnego, nadawany podczas procesu konsolidacji pliku. Jest to informacja wykorzystywana głównie do celów deweloperskich, umożliwiająca zlokalizowanie odpowiednich symboli dodatkowych, wykorzystywanych podczas debugowania programu lub jego profilowania.
- `stripped` – z pliku usunięte są symbole dodatkowe. Najczęściej stosowana metoda zarządzania dodatkowymi symbolami określającymi nazwy funkcji i zmiennych użytych w programie polega na odłączeniu tych symboli od pliku ELF, przeniesieniu ich do zewnętrznego pliku i nadaniu obu plikom tego samego identyfikatora `BuildID`, tak aby można było w przyszłości szybko je ze sobą skojarzyć (np. za pomocą serwera indeksującego). Alternatywną wartością jest `not stripped`, która informuje, że dodatkowe symbole powinny znajdować się bezpośrednio w tym pliku ELF (najczęściej na jego końcu).

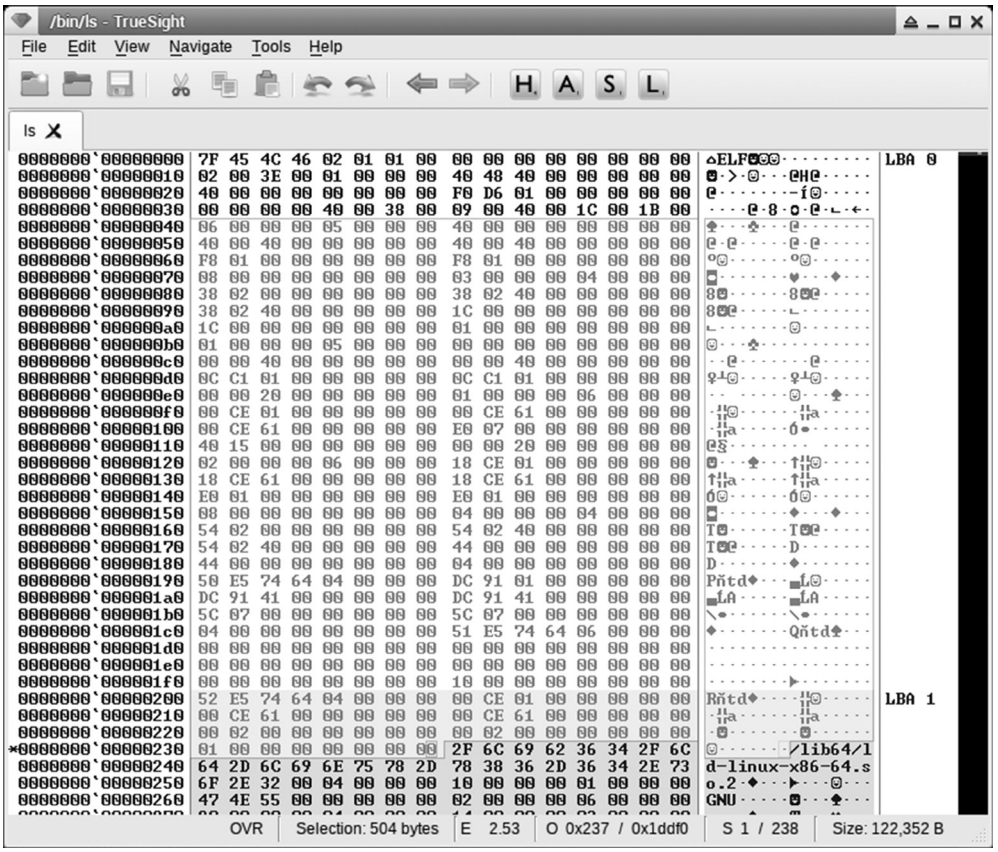
Inne pola w nagłówku *file header* zawierają informacje o adresach i rozmiarach podstawowych struktur formatu ELF. Ich istnienie jest konieczne, ponieważ stanowią punkt wyjściowy dla interpretacji i późniejszego wykonania programu. Będzie o nich mowa w dalszych punktach, przy opisie tych struktur.

2.2.2. Segmenty

Kod wykonywalny, lub dane wykorzystywane przez program, są przechowywane w obszarach noszących nazwę *segmentów*. Granice tych obszarów zdefiniowane są w nagłówku *program header*, który fizycznie znajduje się najczęściej zaraz po wspomnianym w poprzednim punkcie nagłówku *file header*. Dokładny adres nagłówka *program header* jest zapisany w nagłówku *file header*, w polu o adresie 0x1C w przypadku pliku 32-bitowego lub w polu o adresie 0x20 dla pliku 64-bitowego.

Każdy segment opisywany jest przez strukturę `Elf32_Phdr` lub `Elf64_Phdr`. Są one rozmieszczone w pliku w sposób ciągły tak, że opis kolejnego segmentu rozpoczyna się bezpośrednio po zakończeniu struktury opisującej poprzedni segment. Liczba segmentów zadeklarowana jest w polu `e_phnum` w nagłówku *file header* (32-bitowy: adres 0x2C, 64-bitowy: adres 0x38). Na przykład, jeżeli pole `e_phnum` zawiera wartość 9, należy spodziewać się dziewięciu struktur `ElfX_Phdr` występujących bezpośrednio jedna po drugiej.

Na zrzucie ekranu na rysunku 2.2 znajduje się dziewięć segmentów, ulokowanych w pliku w sposób sekwencyjny, jeden po drugim. Każdy segment ma rozmiar odpowiednio 32 lub 56 bajtów (w zależności od architektury 32- lub 64-bitowej). Pokazany tu rozmiar serii nagłówków *program header* to 9 * 56 = 504 bajty.



Rysunek 2.2. Segmenty w pliku ELF

Treść każdego segmentu zawiera informacje takie jak: typ segmentu (pole `p_type`, offset 0x00), jego adres początkowy (pole `p_offset`, czyli pozycja danych w pliku, offset 0x04 lub 0x08), adres w pamięci (pole `p_vaddr`, czyli docelowa pozycja danych w pamięci po załadowaniu pliku, offset 0x08 lub 0x10), jego rozmiary i prawa dostępu do pamięci opisywanej przez ten segment (pole `p_flags`, offset 0x18 dla segmentu 32-bitowego oraz wyjątkowo offset 0x04 dla 64-bitowego).

Istnieje wiele typów segmentów, przeznaczonych do różnych celów. Najważniejsze z nich to:

- Segment `PT_INTERP` (3) – określa obszar przechowujący pełną ścieżkę do specjalnego programu ładującego (nazywanego też „dynamicznym konsolidatorem”,

ang. *dynamic linker*, lub po prostu ld.so), któremu zostanie przekazana kontrola niedługo po procesie wstępnego ładowania pliku ELF przez jądro systemu. Dane, na które wskazuje adres zapisany w polu `p_offset`, zawierają ścieżkę systemu plików.

- Segment `PT_LOAD` (1) – określa obszar danych, które zostaną umieszczone w pamięci pod wybranym adresem. Dane z pliku spod adresu `p_offset` zostaną skopiowane do pamięci pod adres `p_vaddr`, jeśli będzie to możliwe. Jest to typ segmentu zwykle ładujący dane z kodem wykonywalnym programu do pamięci, wraz z dodatkowymi danymi wykorzystywanymi przez ten kod.
- Segment `PT_DYNAMIC` (2) – zawiera informacje dla programu ładującego (określanego przez dane segmentu `PT_INTERP`), wymagane do poprawnego załadowania pliku. Jego istnienie definiuje typ budowy pliku ELF; w przypadku istnienia tego segmentu jest to plik „złączony dynamicznie" (ang. *dynamically linked*), w przypadku braku – plik „złączony statycznie" (ang. *statically linked*).

Należy zwrócić uwagę na to, że kolejność segmentów nie zawsze jest dowolna. Definicje typu `PT_PHDR` lub `PT_INTERP` muszą znajdować się przed definicją `PT_LOAD`. Wynika to ze sposobu ładowania pliku przez system. Może to sugerować interpretację segmentów jako swego rodzaju sekwencję instrukcji do wykonania, które należy zrealizować w określonej kolejności, aby poprawnie załadować plik.

Spośród wymienionych powyżej segmentów, `PT_LOAD` i `PT_DYNAMIC` zawierają najczęściej odczytywane informacje, dlatego poniżej znajduje się ich szerszy opis.

2.2.3. Segment PT_LOAD

Granice obszarów, które mają być załadowane do pamięci, określane są przez dane zdefiniowane w segmencie typu `PT_LOAD`. Program ładujący wykona próbę alokacji pamięci pod adresem wyliczonym na podstawie wartości zapisanej w polu `p_vaddr`. W przypadku powodzenia dane z pliku ELF spod adresu `p_offset` zostaną skopiowane do świeżo przydzielonego regionu pamięci i zostaną tam aż do czasu późniejszej inicjalizacji programu.

Oto przykład wyświetlania informacji o segmentach przy użyciu narzędzia `readelf(1)`:

```
$ readelf -l /bin/ls

Elf file type is EXEC (Executable file)
Entry point 0x404840
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  (...)
```

```
LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
               0x000000000001c10c 0x000000000001c10c  R E    200000
LOAD           0x000000000001ce00 0x000000000061ce00 0x000000000061ce00
               0x00000000000007e0 0x0000000000001540  RW     200000
(...)
```

Często programy wykonywalne, zbudowane w standardowy sposób, mają dwa segmenty typu `PT_LOAD`: pierwszy z nich określa obszar zawierający dane z kodem wykonywalnym, drugi określa dane wykorzystywane przez ten kod. Charakter danych w segmencie można ustalić za pomocą analizy praw dostępu z pola `p_flags`, które określa, jakie prawa dostępu zostaną nadane świeżo przydzielonemu obszarowi pamięci. Jeśli prawa dostępu umożliwiają wykonywanie kodu (flaga `E`), jest to znak, że segment może zawierać kod programu. Z kolei, gdy segment zawiera flagę umożliwiającą zapis do pamięci (flaga `W`), może to być znak, że segment zawiera dane programu, które będą modyfikowane podczas jego działania.

W pokazanym powyżej przykładzie przedstawiony jest plik ELF, w którym zadeklarowano załadowanie dwóch segmentów. Pierwszy z nich to segment ładujący treść pliku z obszaru 0x0 do 0x1C10C, do pamięci pod adres 0x400000. Po załadowaniu pamięci zostanie nadane prawo do wykonania kodu. Faktycznie rozmiar całego obszaru pamięci będzie wynosił 0x1D000, ponieważ rozmiar jest zawsze wyrównywany do rozmiaru strony pamięci zdefiniowanej przez system operacyjny. Drugi segment zawiera deklarację alokacji danych pod adres 0x61CE00. Faktycznie przydzielonym adresem będzie adres 0x61C000, także z powodu wyrównania adresu do rozmiaru strony. Po poprawnej alokacji do adresu 0x61CE00 zostanie skopiowana treść pliku z adresu 0x1CE00. Wypełnienie (od 0x61C000 do 0x61CDFF) poprzedzające alokowany rejon pamięci zostanie nadpisane danymi z pliku, zaczynając od offsetu 0x1C000. Obszar pamięci zostanie oznaczony jako obszar, do którego możliwy jest zapis, ale nie jest możliwe wykonanie z niego kodu.

Istnieją przypadki, w których pole `p_vaddr` nie zawiera preferowanego adresu docelowego. W takich sytuacjach system dobiera ten adres automatycznie. Ta cecha spotykana jest najczęściej w plikach ELF oznaczonych typem `ET_DYN`. Ten typ oryginalnie zarezerwowany był dla bibliotek, a nie aplikacji, jednak zyski płynące z losowości nadania adresu docelowego segmentu spowodowały także wykorzystanie tej techniki przy zwykłych programach, wykonywanych bezpośrednio. Jedną z technik wykorzystujących tę cechę jest ASLR (*Address Space Layout Randomization* [4]).

Możliwy jest też przypadek, gdy zdefiniowany w polu `p_vaddr` adres nie będzie mógł być zaalokowany (sytuacja jest analogiczna do tej, gdy `p_vaddr` będzie zawierało wartość 0x0). Jest to sytuacja problematyczna, ponieważ kompilator podczas generowania kodu deklaruje pewien adres bazowy, pod który kod ten powinien zostać załadowany. Jeśli ten warunek nie zostanie spełniony, kod nie nadaje się do wykonania, ponieważ wszystkie jego referencje do swoich funkcji czy zmiennych są nieprawidłowe; program ładujący zmuszony jest skorzystać wtedy z informacji zawartych w sekcjach `.rela.dyn` oraz `.rela.plt` w celu zmiany położenia (ang. *relocation*, czyli *relokacji*) kodu wykonywalnego pod inny adres. Tylko po modyfikacji kodu zgodnie z informacjami zawartymi w tablicach relokacyjnych możliwe jest wykonanie kodu załadowanego pod inny adres bazowy niż zadeklarowany

podczas procesu kompilacji. Problem ten nie dotyczy kodu, który nie zawiera deklaracji żadnego adresu bazowego, określanego mianem PIC (*Position Independent Code*). Jego odporność na ten problem jest głównym powodem częstego stosowania go w plikach binarnych, w których jest wykorzystany mechanizm ASLR.

2.2.4. Segment PT_DYNAMIC

Jest to specjalny segment, w skład którego wchodzą sekcje odpowiedzialne za przechowywanie informacji o łączeniu dynamicznym. W chwili obecnej standard definiuje tylko jedną taką sekcję: `.dynamic`. Zawiera ona istotnie informacje wykorzystywane przez program ładujący w celu załadowania programu lub biblioteki oraz wszystkich bibliotek zależnych. W przypadku plików wykonywalnych złączonych statycznie, kompilator nie umieszcza w nich informacji o dynamicznym łączeniu i w takich plikach nie ma ani sekcji `.dynamic`, ani segmentu `PT_DYNAMIC`.

2.2.5. Sekcja .dynamic

Każdy program, który nie jest zbudowany statycznie, zawiera sekcję o nazwie `.dynamic`. W przypadku jej istnienia program ładujący musi przejść dodatkowy proces inicjalizacji polegający na załadowaniu każdej biblioteki zależnej oraz – rekurencyjnie – każdej zależności bibliotek zależnych. Jeśli ładowanie jakiejś zależności nie powiedzie się, proces ładowania całego programu kończy się błędem. Treść sekcji `.dynamic`, czyli tabelę z informacjami o łączeniu dynamicznym, można zobaczyć za pomocą programu `readelf(1)`:

```
$ readelf -d /bin/ls

Dynamic section at offset 0x1de18 contains 25 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libcap.so.2]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x4022e8
 0x000000000000000d (FINI)               0x413a4c
 0x0000000000000019 (INIT_ARRAY)         0x61de00
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x61de08
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x400298
 0x0000000000000005 (STRTAB)             0x401060
 0x0000000000000006 (SYMTAB)             0x4003a0
 0x000000000000000a (STRSZ)              1498 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
```