Helion

NIEBIESKI LIS

POLECENIA **PROCESORÓW ARM** I INŻYNIERIA WSTECZNA

MARIA AZERIA MARKSTEDTER

WILEY

Tytuł oryginału: Blue Fox: Arm Assembly Internals and Reverse Engineering

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-289-0675-4

Copyright © 2023 by John Wiley & Sons, Inc.

All Rights Reserved. This translation published under license with the original publisher John Wiley & Sons, Inc.

Translation copyright © 2024 by Helion S.A.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, without either the prior written permission of the Publisher.

WILEY and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/nielis
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

O autorce 13

Podziękowania 14

Wprowadzenie 15

Część I. Asembler procesora Arm 19

Rozdział 1. Wprowadzenie do inżynierii wstecznej 21
Wprowadzenie do asemblera 21
Bity i bajty 22
Kodowanie znaków 23
Kod maszynowy i asembler 24
Asemblacja 26
Asemblery skrośne 30
Języki wysokiego poziomu 31
Dezasemblacja 32
Dekompilacja 33

Rozdział 2. Właściwości formatu plików ELF 36
Struktura programu 36
Języki wysokiego poziomu a języki niskiego poziomu 37
Proces kompilacji 39
Kompilacja pod kątem różnych architektur 40
Asemblacja i linkowanie 41
Struktura pliku ELF 44

Kup ksi k
Pole ksi k

Nagłówek pliku ELF 44
Pola informacyjne nagłówka pliku ELF 45
Pola platformy docelowej 46
Pole punktu wejściowego 46
Pola lokalizacji tabel 47
Nagłówki programu ELF 47
Nagłówek programu PHDR 48
Nagłówek programu INTERP 48
Nagłówki programu LOAD 49
Nagłówek programu DYNAMIC 50
Nagłówek programu NOTE 50
Nagłówek programu TLS 50
Nagłówek programu GNU_EH_FRAME 51
Nagłówek programu GNU_STACK 51
Nagłówek programu GNU_RELRO 53
Nagłówki sekcji pliku ELF 54
Metasekcje pliku ELF 56
Sekcja tabeli łańcuchów 56
Sekcja tabeli symboli 57
Główne sekcje pliku ELF 57
Sekcja .text 57
Sekcja .data 57
Sekcja .bss 57
Sekcja .rodata 58
Sekcje .tdata i .tbss 58
Symbole 58
Symbole globalne i lokalne 60
Symbole słabe 60
Wersje symboli 60
Symbole mapowania 61
Sekcja dynamiczna i ładowanie dynamiczne 61
Ładowanie zależności (NEEDED) 62
Relokacje programu 63
Relokacje statyczne 63
Relokacje dynamiczne 64
Globalna tabela przesunięć (GOT) 65
Tabela łączenia procedur (PLT) 65
Sekcje inicjalizacji i terminacji programu 66
Kolejność inicjalizacji i terminacji 68
Pamięć lokalna wątków 68
Model dostępu local-exec 72
Model dostępu TLS initial-exec 72
Model dostępu TLS general-dynamic 73
Model dostępu TLS local-dynamic 75

Rozdział 3. Podstawy systemu operacyjnego 76
Ogólna architektura systemu operacyjnego 76
Tryb użytkownika a tryb jądra 76
Procesy 77
Wywołania systemowe 78
Obiekty i uchwyty 83
Wątki 85
Zarządzanie pamięcią procesu 86
Strony pamięci 87
Zabezpieczenia pamięci 88
Pamięć anonimowa i mapowana 89
Pliki i moduły mapowane w pamięci 89
Randomizacja układu przestrzeni adresowej 92
Implementacje stosu 94
Pamięć współdzielona 96

Rozdział 4. Architektura Arm 97
Architektury i profile 97
Architektura Armv8-A 99
Poziomy wyjątków 99
Rozszerzenie Armv8-A TrustZone 100
Zmiany poziomu wyjątków 102
Stany wykonywania Armv8-A 104
Stan wykonywania AArch64 106
Zestaw rozkazów A64 106
Rejestry AArch64 107
Licznik programu 109
Wskaźnik stosu 109
Rejestr zerowy 110
Rejestr łączenia 111
Wskaźnik ramki 111
Rejestr platformy (x18) 111
Rejestry wywołań międzyproceduralnych 112
Rejestry SIMD i zmiennoprzecinkowe 112
Rejestry systemowe 113
PSTATE 114
Stan wykonywania AArch32 115
Zestawy rozkazów A32 i T32 116
Zestaw rozkazów A32 116
Zestaw rozkazów T32 116
Przełączanie między zestawami rozkazów 117

Rejestry AArch32 120
Licznik programu 120
Wskaźnik stosu 121
Wskaźnik ramki 121
Rejestr połączenia 121
Rejestr wywołań międzyproceduralnych (IP, r12) 122
Rejestr bieżącego stanu programu 122
Rejestr stanu programu 122
Rejestry stanu wykonywania 124
Rejestr stanu zestawu rozkazów 125
Rejestr stanu bloku IT (ITSTATE) 125
Stan kolejności bitów 126
Bity trybu i maski wyjątków 126

Rozdział 5. Rozkazy przetwarzania danych 129
Operacje przesunięcia i obrotu 131
Przesunięcie logiczne w lewo 132
Przesunięcie logiczne w prawo 132
Przesunięcie arytmetyczne w prawo 133
Obrót w prawo 133
Obrót w prawo z przeniesieniem 134
Formy rozkazów 134
Forma przesunięcia o bezpośrednio podaną wartość stałą 135
Forma przesunięcia o wartość zapisaną w rejestrze 137
Operacje manipulacji bitami 138
Przesunięcie pola bitowego 139
Operacje rozszerzenia znakiem lub zerami 143
Pobieranie i wstawianie pól bitowych 147
Operacje logiczne 149
Binarna operacja AND 149
Rozkaz TST 150
Binarne czyszczenie bitów 151
Binarna operacja OR 151
Binarna operacja OR NOT 152
Binarna operacja OR wykluczającego 153
Rozkaz TEQ 154
Binarna operacja OR NOT wykluczającego 154
Operacje arytmetyczne 155
Dodawanie i odejmowanie 155
Odejmowanie odwrotne 157
Porównywanie 157
Rozkaz CMN 158
Operacje mnożenia 160

Operacje mnożenia w zestawie A64 160
Operacje mnożenia w zestawach A32/T32 162
Operacje mnożenia słów wykorzystujące najmłodsze bity 163
Operacje mnożenia słów wykorzystujące najstarsze bity 164
Operacje mnożenia półsłów 166
Operacje mnożenia wektorów (podwójne) 169
Operacje mnożenia wartości długich (64-bitowych) 172
Operacje dzielenia 178
Operacje przeniesienia 179
Przeniesienie stałej wartości bezpośredniej 180
Przeniesienie wartości bezpośredniej i MOVT w zestawach A32/T32 180
Przeniesienie wartości bezpośredniej (MOVZ) i MOVK w zestawie A64 181
Przeniesienie rejestru 182
Przeniesienie z negacją 184

Rozdział 6. Rozkazy dostępu do pamięci 185
Podstawowe informacje 185
Tryby adresowania i formy offsetu 187
Adresowanie z offsetem 190
Stała wartość bezpośrednia jako offset 191
Offsety rejestrowe 196
Tryb preindeksowany 198
Przykład użycia trybu preindeksowania 198
Adresowanie postindeksowane 200
Przykład użycia adresowania postindeksowanego 201
Adresowanie literałowe (względem PC) 202
Ładowanie stałych 202
Ładowanie adresu do rejestru 205
Rozkazy ładowania i zapisu 208
Ładowanie i zapis słów i podwójnych słów 208
Ładowanie i zapis półsłów lub bajtów 210
Przykład użycia rozkazów ładowania i zapisu 213
Ładowanie i zapis wielu wartości (A32) 214
Przykład użycia rozkazów STM i LDM 221
Bardziej skomplikowany przykład użycia rozkazów STM i LDM 221
Ładowanie i zapis par (A64) 223

Rozdział 7. Wykonywanie warunkowe 227
Wykonywanie warunkowe — informacje ogólne 227
Kody warunkowe 228
Flagi warunkowe NZCV 228
Przepełnienie całkowitoliczbowe ze znakiem i bez znaku 229
Kody warunkowe 231

Rozkazy warunkowe 232
Rozkaz IT w Thumb 233
Rozkazy ustawiające flagi 235
Przyrostek rozkazu S 235
Przyrostek S w rozkazach dodawania i odejmowania 236
Przyrostek S w rozkazach przesunięcia logicznego 238
Przyrostek S w rozkazach mnożenia 239
Przyrostek S w innych rozkazach 239
Rozkazy testowania i porównywania 240
Porównywanie (CMP) 240
Porównywanie z negacją (CMN) 242
Testowanie bitów (TST) 243
Sprawdzanie pod kątem równości (TEQ) 246
Rozkazy wyboru warunkowego 247
Rozkazy porównywania warunkowego 250
Operacje warunkowe z logicznym AND i rozkazem CCMP 250
Operacje warunkowe z logicznym OR i rozkazem CCMP 253

Rozdział 8. Kontrola przepływu sterowania 256
Rozkazy rozgałęziające 256
Rozgałęzienia warunkowe i pętle 257
Rozgałęzienia z testem i porównaniem 261
Rozgałęzienia tabelowe 262
Rozgałęzienie i zamiana 264
Rozgałęzienia do podprocedur 267
Funkcje i podprocedury 269
Standard wywoływania procedur 270
Rejestry ulotne i nieulotne 271
Argumenty i wartości zwrotne 272
Przekazywanie większych wartości 274
Funkcje liście i funkcje niebędące liśćmi 276
Funkcje liście 277
Funkcje niebędące liśćmi 277
Prolog i epilog 277

Część II. Inżynieria wsteczna 283

Rozdział 9. Środowiska Arm 285
Płytki Arm 286
Emulator QEMU 287
Emulacja w trybie użytkownika QEMU 288
Pełna emulacja systemu w QEMU 291
Emulacja oprogramowania układowego 292

Rozdział 10. Analiza statyczna 297
Narzędzia do analizy statycznej 298
Narzędzia wiersza poleceń 298
Dezasemblery i dekompilatory 298
Binary Ninja Cloud 299
Przykład analizy wywołania przez referencję 304
Analiza przepływu sterowania 309
Funkcja main 310
Podprocedura 311
Konwersja na char 315
Instrukcja if 316
Dzielenie współczynnika 318
Pętla for 320
Analiza algorytmu 322

Rozdział 11. Analiza dynamiczna 335
Debugowanie w wierszu poleceń 336
Polecenia GDB 337
Konfiguracja GDB 338
Rozszerzenie GEF debugera GDB 340
Instalacja 340
Interfejs 341
Przydatne polecenia GEF 341
Badanie pamięci 344
Oglądanie obszarów pamięci 346
Analizatory luk bezpieczeństwa 347
Polecenie checksec 349
Radare2 351
Debugowanie 351
Debugowanie zdalne 356
Radare2 356
IDA Pro 356
Debugowanie błędu pamięci 357
Debugowanie procesu w GDB 365

Rozdział 12. Inżynieria wsteczna programów arm64 w systemie macOS 370
Podstawowe informacje 371
Pliki binarne arm64 dla systemu macOS 372
Program powitalny dla systemu macOS (arm64) 375
Polowanie na złośliwe pliki binarne arm64 377

Kup ksi k Pole ksi k

Analiza złośliwego oprogramowania arm64 383
Techniki utrudniające analizę 384
Logika utrudniająca debugowanie (przez ptrace) 385
Logika utrudniająca debugowanie (przez sysctl) 388
Zabezpieczenie przed maszynami wirtualnymi (stan SIP i wykrywanie artefaktów maszyny wirtualnej) 391
Podsumowanie 396

Skorowidz 399

Kup ksi k Pole ksi k

Rozdział

10

Analiza statyczna

W części I omówiłam najczęściej używane rozkazy, jakie można napotkać podczas dezasemblacji programów. Teraz zastosujemy tę wiedzę w praktyce i nauczysz się analizować działanie programów w postaci plików binarnych. Przykłady prezentowane w tym rozdziale są proste i zrozumiałe, a dzięki ich szczegółowej analizie utrwalisz zdobytą do tej pory wiedzę.

Czym jest **analiza statyczna**? Pojęcie to ma różne znaczenia w zależności od tego, kogo spytamy, ale jedna cecha będzie wspólna u wszystkich: jest to analiza pliku w jego statycznej postaci, bez uruchamiania. W tym rozdziale analiza statyczna oznacza niskopoziomowe badanie pliku binarnego.

Analiza statyczna poprzedza analizę dynamiczną, ponieważ aby zrozumieć działanie programu, należy najpierw zapoznać się z jego podstawowymi właściwościami. Musimy przecież wiedzieć, jakich zasobów i środowiska wymaga do działania. Lekka analiza statyczna pozwala przygotować odpowiednie środowisko i narzędzia do analizy pliku na podstawie jego typu oraz zrozumieć jego strukturę na podstawie formatu.

Samo zgromadzenie informacji na temat podstawowych właściwości pliku często jest niewystarczające, aby przejść do fazy analizy dynamicznej. W takich przypadkach należy zidentyfikować w kodzie punkty, w których chcemy obserwować jego interakcję z systemem w celu uzyskania głębszej wiedzy na temat jego funkcjonalności. Na przykład jeśli złośliwy program wykonuje zadania sieciowe, deszyfruje dane lub modyfikuje pliki w systemie plików, musimy wiedzieć, gdzie szukać i które strumienie danych monitorować w trakcie jego działania.

To wymaga analizy programu po dezasemblacji, która pozwala na zrozumienie sposobu jego działania. W ten sposób można nie tylko dowiedzieć się, co robią poszczególne funkcje, ale także można odkryć funkcje, które są uruchamiane tylko w określonych warunkach.

Dla analityka luk bezpieczeństwa umiejętność wykonywania niskopoziomowej analizy jest podstawową umiejętnością. Analiza zdezasemblowanego kodu funkcji zawierającej lukę bezpieczeństwa może pomóc w zrozumieniu, w jakich warunkach ta funkcja jest uruchamiana oraz jaki dokładnie strumień danych jest wymagany, aby ją wykorzystać bez spowodowania awarii programu.

Kup ksi k

Pole ksi k

Narzędzia do analizy statycznej

W tym podrozdziale znajduje się zwięzły przegląd narzędzi do analizy statycznej, których można używać do inżynierii wstecznej. W zależności od potrzeb i systemu operacyjnego konieczne może być użycie ich kombinacji z programami z GUI. Narzędzia wiersza poleceń umożliwiają wykonywanie lekkiej analizy statycznej, zbieranie ogólnych informacji o plikach binarnych, które chcemy poddać inżynierii wstecznej, oraz szybką analizę mniejszych plików binarnych. Dezasemblery i dekompilatory z GUI to zaawansowane narzędzia, których można używać z rozszerzeniami i własnymi skryptami.

Narzędzia wiersza poleceń

Narzędzia wiersza poleceń mogą być bardzo przydatne podczas procesów inżynierii wstecznej, zwłaszcza we wstępnej fazie zbierania informacji. Warto znać takie linuksowe polecenia jak `strings`, które zwraca listę wszystkich łańcuchów znajdujących się w pliku w kolejności występowania, `file`, które zwraca typ pliku, czy `readelf`, które zwraca cenne informacje na temat plików w formacie ELF.

Aby przeprowadzić dezasemblację pliku w Linuksie, również wystarczy tylko jedno polecenie. Opcja `-d` polecenia `objdump` zwraca kod dezasemblera każdej funkcji znajdującej się w sekcjach kodu wykonywalnego. Narzędzia wiersza poleceń przydają się jednak nie tylko do lekkiej analizy statycznej. Za pomocą programu GDB można na przykład debugować program z poziomu wiersza poleceń, a nawet korzystać z bardziej zaawansowanych narzędzi, takich jak Radare2.

Dezasemblery i dekompilatory

Dezasemblery pozwalają obejrzeć niskopoziomowy kod programu i są dostępne w różnych odmianach i cenach, od darmowych narzędzi open source, takich jak Radare2 i Ghidra, po komercyjne programy, takie jak Binary Ninja i IDA Pro. Niektóre z nich mają funkcje dekompilacji, które odtwarzają kod wysokopoziomowy zdezasemblowanego programu. Oto lista popularnych dezasemblerów i dekompilatorów:

- **IDA Pro**[1] to zaawansowany dezasembler i debuger, a jednocześnie najdroższa opcja na rynku. Obsługuje wiele architektur procesorów, umożliwia przedstawienie bloków kodu i przepływu sterowania między nimi w postaci grafu oraz pozwala tworzyć wtyczki w postaci skryptów. Hex-Rays to dekompilator C i C++, który można kupić jako wtyczkę do IDA Pro.
- **Binary Ninja**[2] to interaktywny dezasembler, dekompilator i platforma do analizy binarnej w znacznie przystępniejszej cenie niż IDA Pro. Ma wiele funkcji, w tym dezasembler, dekompilator, automatyzację poprzez zaawansowane API, widoki języka pośredniego, a nawet dezasembler chmurowy[3]. Dekompilator tego narzędzia jest wyjątkowy. Wykorzystuje architekturę bazującą na drzewie i pośrednią reprezentację

[1] *hex-rays.com/IDA-pro.*

[2] *binary.ninja.*

[3] *cloud.binary.ninja.*

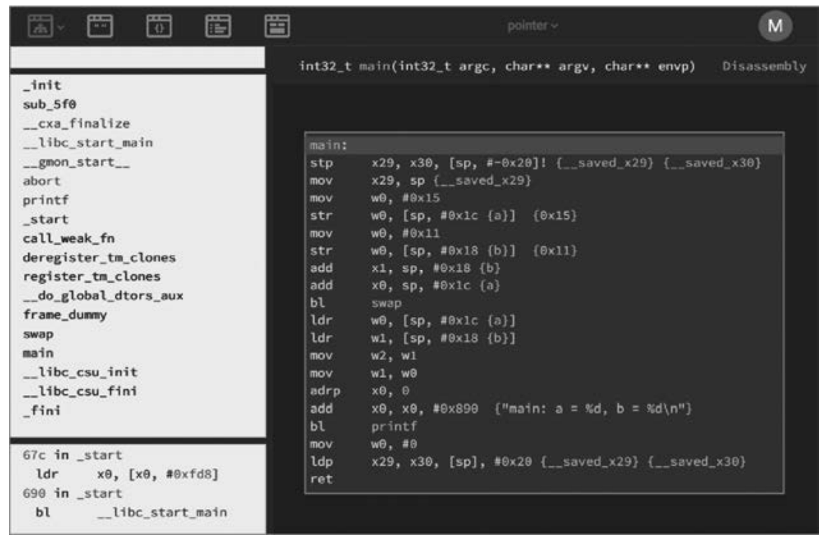
kodu maszynowego o nazwie Binary Ninja Intermediate Language (BNIL)[4] oraz może wyświetlać zdezasemblowany kod na trzech różnych poziomach abstrakcji: język pośredni niskiego poziomu, język pośredni średniego poziomu, język pośredni wysokiego poziomu. Przydaje się to w szczególności użytkownikom, którzy w trakcie analizy chcą zachować część szczegółów dostarczanych przez język asemblera.

- **Ghidra** to otwartoźródłowy pakiet narzędzi do inżynierii wstecznej utworzony przez Research Directorate NSA[5]. Ma funkcje dezasemblera, asemblera, dekompilatora, debugera oraz interpretera skryptów.
- Narzędzie open source **Radare2**[6] to zaawansowany dezasembler i debuger działający w wierszu poleceń. Ma wiele funkcji do analizy binarnej i inżynierii wstecznej.

Binary Ninja Cloud

Binary Ninja to zaawansowane narzędzie do inżynierii wstecznej, które należy do moich ulubionych. Ma wyjątkowe funkcje i jest dostępne w wersji chmurowej, na którą warto zwrócić uwagę. Program Binary Ninja Cloud jest bardzo pomocny, gdy nie mamy dostępu do swojego dezasemblera używanego na co dzień albo chcemy wygodnie wykonywać działania z zakresu inżynierii wstecznej w oknie przeglądarki. Przyjrzyjmy się jego funkcjom.

Kiedy prześlemy plik binarny, zostaje nam zaprezentowana lista funkcji i graf dezasemblera. Na rysunku 10.1 widać funkcję `main`.



Rysunek 10.1. Funkcja main w Binary Ninja

4 *docs.binary.ninja/dev/bnil-overview.html.*

5 *github.com/NationalSecurityAgency/ghidra.*

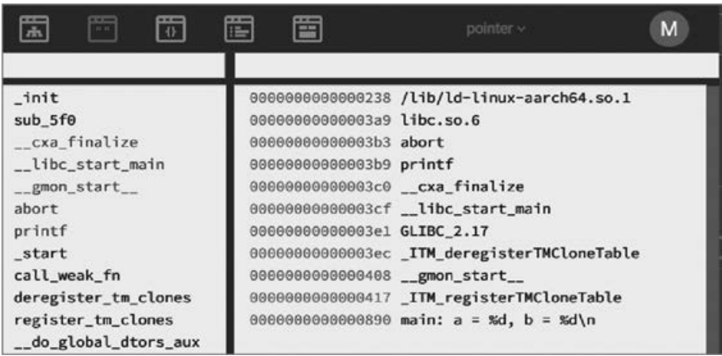
6 *rada.re/n/radare2.html.*

Kup książkę Poleć książkę

Dodatkowo możemy włączyć pokazywanie adresów, bajtów kodów operacyjnych i typów zmiennych (zobacz rysunek 10.2).

Rysunek 10.2. Opcje wyświetlania Binary Ninja

Ponadto Binary Ninja może wyświetlić łańcuchy pliku binarnego (zobacz rysunek 10.3).



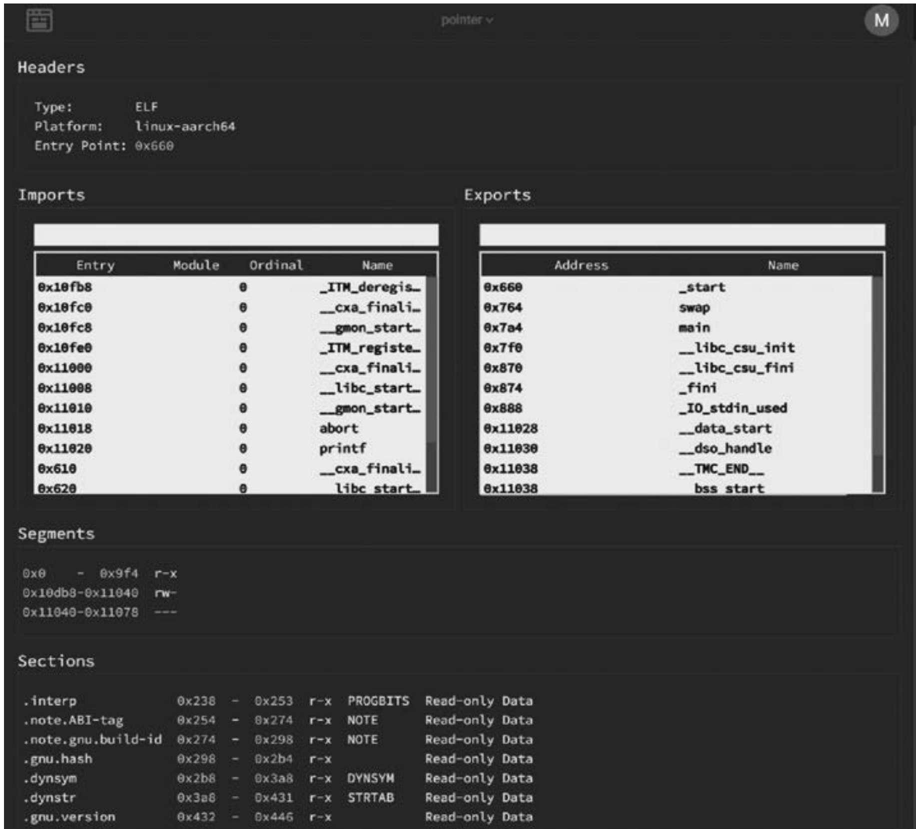
Rysunek 10.3. Wyświetlanie łańcuchów

Na rysunku 10.4 jest pokazana funkcja Triage, która dostarcza informacje o plikach binarnych, np. o nagłówkach, importach i eksportach, segmentach oraz sekcjach.

Program Binary Ninja ma kilka wyjątkowych funkcji dekompilacji, które warto poznać. Jak napisałam wcześniej, narzędzie to umożliwia prezentację zdezasemblowanego kodu na różnych poziomach abstrakcji, w formie niskopoziomowego, średniopoziomowego i wysokopoziomowego języka pośredniego.

Inne dezasemblery pokazują tylko nieprzetworzony zdezasemblowany kod programu i ewentualnie wysokopoziomowy pseudokod, jeśli są wyposażone w funkcję dekompilacji. Jednak w zależności od rodzaju analizy, jaką chcemy przeprowadzić, nadal potrzebujemy pewnego poziomu szczegółowości, choć może nie aż tak dużego, jak odczyt każdego rozkazu asemblera. Jeśli chcemy sprawdzić,

Kup ksi k Pole ksi k



Rysunek 10.4. Funkcja Triage

jak zmieniają się wybrane rejestry i lokalizacje w pamięci bez analizowania każdego rozkazu, to przydatny może się okazać język pośredni niskiego poziomu. A jeśli nie potrzebujemy aż takich szczegółów i chcemy tylko poznać wynik dla wybranych rejestrów, zanim zostaną użyte w wywołaniu funkcji, to przydatny może być język pośredni średniego poziomu. Jeżeli natomiast zależy nam na wysokopoziomowej perspektywie w postaci pseudokodu, to użyjemy języka pośredniego wysokiego poziomu.

Przyjrzyjmy się funkcji `main` w języku pośrednim niskiego poziomu. Na rysunku 10.5. widać, jak zmienił się poziom abstrakcji. Nadal jest to poziom szczegółowości kodu asemblera. Każdy wiersz pokazuje, jak zmieniają się rejestry w bardziej czytelny sposób, bez podawania mnemoników rozkazów. W pierwszej chwili może to wydawać się niejasne, ale zwróć uwagę, że w każdym wierszu mamy informację na temat tego, jak zmieniają się rejestry i zawartość pamięci. Na przykład wiersz 1 pokazuje, że wartość `SP` jest zmniejszana o `0x20`. W wierszu 2 widzimy, że wartość rejestru `x29` zostaje zapisana na stosie pod adresem w `SP` powiększonym o offset o etykiecie `__saved_x29` itd.

Dzięki temu kod asemblera jest czytelniejszy, ale co, jeśli nie interesuje nas, jak zmienia się każdy rejestr na każdym kroku? Kod w języku pośrednim średniego poziomu przedstawiony na rysunku 10.6 jest łatwiejszy do zrozumienia.

```
int32_t main(int32_t argc, char** argv, char** envp)    LLIL

main:
sp = sp - 0x20
[sp {__saved_x29}].q = x29
[sp + 8 {__saved_x30}].q = x30
x29 = sp {__saved_x29}
w0 = 0x15
[sp + 0x1c {a}].d = w0
w0 = 0x11
[sp + 0x18 {b}].d = w0
x1 = sp + 0x18 {b}
x0 = sp + 0x1c {a}
call(swap)
w0 = [sp + 0x1c {a}].d
w1 = [sp + 0x18 {b}].d
w2 = w1
w1 = w0
x0 = __elf_header
x0 = x0 + 0x890  {"main: a = %d, b = %d\n"}
call(printf)
w0 = 0
x29 = [sp {__saved_x29}].q
x30 = [sp + 8 {__saved_x30}].q
sp = sp + 0x20
<return> jump(x30)
```

Rysunek 10.5. Niskopoziomowy język pośredni

```
int32_t main(int32_t argc, char** argv, char** envp)    MLIL

main:
a = 0x15
b = 0x11
x1 = &b
x0 = &a
swap(x0, x1)
x0_1 = a
x1_1 = b
x2 = zx.q(x1_1)
x1_2 = zx.q(x0_1)
printf("main: a = %d, b = %d\n", x1_2, x2)
x0_2 = 0
return 0
```

Rysunek 10.6. Język pośredni średniego poziomu

Zwróć uwagę, że kod został zredukowany do najważniejszych elementów. Nie widzimy takich szczegółów jak to, gdzie na stosie są przechowywane wartości zmiennych, które nie są istotne w naszej analizie. W tym widoku możemy obejrzeć wartości zmiennych `a` i `b`, a co najważniejsze, rejestry argumentów `X0` i `X1` zostają zapełnione adresami `a` i `b`.

Jeśli ustawimy język pośredni o poziom wyżej, na poziom wysoki, to otrzymamy zdekompilowaną wersję pierwotnego kodu źródłowego (zobacz rysunek 10.7).

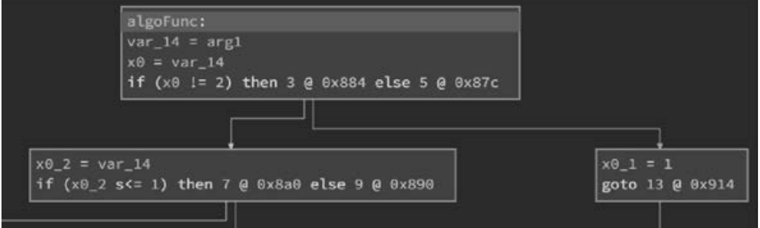
```
int32_t main(int32_t argc, char** argv, char** envp)    HLIL

main:
int32_t a = 0x15
int32_t b = 0x11
swap(&a, &b)
printf("main: a = %d, b = %d\n", zx.q(a), zx.q(b))
return 0
```

Rysunek 10.7. Język pośredni wysokiego poziomu

Warunkowy przepływ sterowania programu także można uprościć do bardziej czytelnej postaci. Na rysunku 10.8 widać wersję warunkowego przepływu sterowania w asemblerze. W widoku kodu pośredniego średniego poziomu ten sam kod został zredukowany do podstawowej logiki i stał się znacznie bardziej czytelny (zobacz rysunek 10.9).

```
algoFunc:
stp     x29, x30, [sp, #-0x30]! {__saved_x29} {__saved_x30}
mov     x29, sp {__saved_x29}
str     w0, [sp, #0x1c {var_14}]
ldr     w0, [sp, #0x1c {var_14}]
cmp     w0, #0x2
b.ne    0x884

ldr     w0, [sp, #0x1c {var_14}]
cmp     w0, #0x1
b.le    0x8a0

mov     w0, #0x1
b       0x910
```

Rysunek 10.8. Widok wykresu przepływu warunkowego



Rysunek 10.9. Widok grafu języka pośredniego średniego poziomu

Mimo że takie narzędzia jak Binary Ninja mogą uprościć analizę, pamiętaj, że dokładna znajomość rozkazów asemblera wciąż jest podstawą dla każdego specjalisty od inżynierii wstecznej. Dlatego poniżej opisuję techniki inżynierii wstecznej struktur sterowania przepływem i funkcji na poziomie asemblera. Teraz przyjrzymy się każdemu rozkazowi, aby lepiej zrozumieć zasadę działania wskaźników, nauczyć się analizować warunkowy przepływ sterowania programu oraz poćwiczyć czytanie kodu asemblera poprzez analizę algorytmu od początku. Kod asemblera przedstawiony na rysunkach w tym podrozdziale pochodzi z dezasemblera IDA Pro.

Kup ksi k Pole ksi k

Przykład analizy wywołania przez referencję

Osobom uczącym się programowania w języku C często problemy sprawiają wskaźniki. Niezależnie od tego, czy jesteś doświadczonym programistą C, czy dopiero się uczysz programowania w tym języku, analiza operacji wskaźnikowych z poziomu asemblera pozwoli Ci dokładniej zrozumieć ich działanie.

Zaczniemy od przykładu, który często spotyka się podczas inżynierii wstecznej — od analizy wywołania przez referencję. Ta metoda wywoływania funkcji polega na przekazaniu wartości przez referencję, czyli przekazaniu jej adresu do funkcji, która przeprowadza dereferencję skopiowanego adresu w celu uzyskania dostępu do obiektów. Innymi słowy, zamiast samej wartości do funkcji jest przekazywany jej adres.

Kiedy deklarujemy zmienne w funkcji głównej i przekazujemy ich wartości do innej funkcji, to ta druga funkcja pobiera tylko kopie tych wartości jako argumenty i nie może zmodyfikować oryginalnych zmiennych:

```
#include <stdio.h>

void swap(int a, int b){
    int t = a;
    a = b;
    b = t;
    printf("W swap: a = %d, b = %d\n", a, b);
    return;
}

int main(void) {

    int a = 21;
    int b = 17;
    printf("Przed swap: a = %d, b = %d\n", a, b);
    swap(a, b);
    printf("Poza swap: a = %d, b = %d\n", a, b);
    return 0;
}

Wynik:
Przed swap: a = 21, b = 17
W swap: a = 17, b = 21
Poza swap: a = 21, b = 17
```

W przypadku wywołania przez referencję jako argumenty funkcji przekazujemy adresy obiektów (`&a, &b`). Funkcja deklaruje te argumenty jako wskaźniki do wartości typu `int` (`int *pa, int *pb`) i przeprowadza dereferencję skopiowanych adresów, aby uzyskać dostęp do oryginalnych obiektów.

Innymi słowy, kiedy jako argument funkcji przekazujemy adres zmiennej, to funkcja ta wykonuje operacje na wartościach przechowywanych pod adresem oryginalnej zmiennej:

```
#include <stdio.h>

void swap(int *pa, int *pb){
    int t = *pa;
    *pa = *pb;
    *pb = t;
    return;
}
```

```
int main(void) {

    int a = 21;
    int b = 17;

    swap(&a, &b);
    printf("main: a = %d, b = %d\n", a, b);

    return 0;
}

Wynik:
main: a = 17, b = 21
```

Uwaga **W języku C operator referencji & oznacza „adres”, a operator dereferencji * oznacza „wartość wskazywana przez”.**

Zaczniemy od funkcji głównej. Najpierw zmienne a i b zostają zainicjalizowane odpowiednio wartościami 21 i 17. W kodzie asemblera przekłada się to na inicjalizację rejestru W0 wartością zmiennej i zapisanie jej w odpowiednim miejscu na stosie. W programie IDA offset tej lokalizacji ma etykietę odpowiadającą zmiennej. Na rysunku 10.10 nazwy tych etykiet odpowiednio zmieniłam dla zwiększenia czytelności.

```
                        main:
                        var_20 = -0x20
                        b = -8
                        a = -4

                          STP  X29, X30, [SP, #var_20]!
int main(void) {          MOV  X29, SP
                          MOV  W0, #0x15
    int a = 21; ------->  STR  W0, [SP,#0x20+a]
                          MOV  W0, #0x11
    int b = 17; ------->  STR  W0, [SP,#0x20+b]
    ...
```

Rysunek 10.10. Inicjalizacja zmiennych a i b

Funkcja swap pobiera adresy zmiennych a i b jako argumenty funkcji. Oznacza to, że do rejestrów argumentów X0 i X1 muszą zostać wstawione adresy zmiennych a i b, a nie wartości przechowywane pod tymi adresami. Na rysunku 10.11 widać, że adresy lokalizacji tych zmiennych na stosie zostały wstawione do rejestrów X0 i X1 za pomocą rozkazów ADD.

```
                          main:
                            ...
                            MOV  W0, #0x15
int main(void) {            STR  W0, [SP,#0x20+a]
                            MOV  W0, #0x11
    int a = 21;             STR  W0, [SP,#0x20+b]
    int b = 17;             ADD  X1, SP, #0x20+b
    swap(&a, &b); ------->  ADD  X0, SP, #0x20+a
                            BL   swap
```

Rysunek 10.11. Przygotowanie argumentów dla funkcji swap

Parametry funkcji swap są zadeklarowane jako wskaźniki do zmiennych typu int. Ponieważ adresy zmiennych a i b zostały przekazane jako argumenty funkcji, pa i pb zawierają teraz kopie tych adresów i odnoszą się do tych samych obiektów.

Innymi słowy, pa i pb zawierają adresy, a więc wskazują zawartość zmiennych a i b. To znaczy, że możemy ją pobrać przez dereferencję (*) wskaźników pa i pb:

```
Pa = 0x0000ffffffff43c
  *pa = 21
pb = 0x0000ffffffff438
  *pb = 17
t = 21

[SP,#0x20+pa] = 0x0000ffffffff43c -> 21
[SP,#0x20+pb] = 0x0000ffffffff438 -> 17
```

Na rysunku 10.12 widać, że adresy przekazywane przez rejestry argumentów X0 i X1 są przechowywane w przeznaczonych dla nich lokalizacjach na stosie. Miejsce na stosie o adresie [SP,#0x20+pa] zawiera teraz adres do wartości zmiennej a, natomiast miejsce o adresie [SP,#0x20+pb] zawiera adres do wartości zmiennej b.

```
void swap(int *pa, int *pb){
    int t = *pa;
    ...
}
```

```
swap:
    SUB    SP, SP, #0x20
    STR    X0, [SP,#0x20+pa]
    STR    X1, [SP,#0x20+pb]
    LDR    X0, [SP,#0x20+pa]
    LDR    W0, [X0]
    STR    W0, [SP,#0x20+t]
```

Rysunek 10.12. Wywołanie funkcji swap i zapis argumentów na stosie. Zmienna t jest zainicjalizowana

Pierwszy wiersz funkcji swap inicjalizuje zmienną t wartością wskazywaną przez pa. W kodzie asemblera pierwszy rozkaz LDR ładuje *adres* przechowywany w [SP,#0x20+pa] do rejestru X0:

```
$x0 : 0x0000ffffffff43c → 0x0000000000000015
```

Drugi rozkaz LDR ładuje *wartość* spod tego adresu do W0:

```
$x0 : 0x15
```

Aby zainicjalizować zmienną t tą wartością, rozkaz STR zapisuje wcześniej załadowaną wartość w lokalizacji na stosie przeznaczonej dla tej zmiennej. Rysunek 10.13 przedstawia abstrakcyjną ilustrację lokalizacji zmiennych na stosie i ich zawartości.

```
[SP,#0x20+a]  |: | 0x15              |
[SP,#0x20+b]  |: | 0x11              |
[SP,#0x20+pa] |: | 0000ffffffff43c   | -> [a]: 0x15 (21)
[SP,#0x20+pb] |: | 0000ffffffff438   | -> [b]: 0x11 (17)
[SP,#0x20+t]  |: | 0x15              |
```

Rysunek 10.13. Lokalizacje zmiennych na stosie

Drugi wiersz w funkcji swap zmienia wartość wskazywaną przez pa na wartość wskazywaną przez pb. Pamiętaj, że dokonujemy dereferencji wskaźników, aby zmodyfikować wartość, którą

wskazują, a nie same adresy. Oznacza to, że pa i pb nadal zawierają ten sam adres, ale wartość wskazywana przez pa uległa zmianie.

```
Pa = 0x0000ffffffff43c
    *pa = 17
pb = 0x0000ffffffff438
    *pb = 17
t = 21
```

Na rysunku 10.14 widać, że program przeprowadza dereferencję wskaźników w dwóch krokach. Jako że adres wartości jest przechowywany w [SP,#0x20+pb], program najpierw ładuje ten adres do rejestru X0, a następnie używa tego rejestru jako adresu źródłowego do załadowania wartości do rejestru W1. Rejestrem docelowym jest W1, ponieważ wartość jest przechowywana w 32 pierwszych bitach lokalizacji na stosie. Po wykonaniu dwóch pierwszych rozkazów rejestry X0 i X1 zawierają następujące wartości:

```
$x0 : 0x0000ffffffff438 → 0x0000001500000011
$x1 : 0x11
```

```
void swap(int *pa, int *pb){          swap:
    int t = *pa;                         ...
    *pa = *pb;           ———→           LDR    X0, [SP,#0x20+pb]
    ...                                  LDR    W1, [X0]
}                                        LDR    X0, [SP,#0x20+pa]
                                         STR    W1, [X0]
```

Rysunek 10.14. Dereferencja wskaźników oraz ustawienie wartości *pa na wartość *pb

Aby zmienić wartość wskazywaną przez pa, program najpierw ładuje adres zlokalizowany w [SP,#0x20+pa] do X0, a następnie używa go jako adresu docelowego do zapisania wartości W1. Po wykonaniu ostatnich dwóch rozkazów rejestry X0 i X1 zawierają następujące wartości:

```
$x0 : 0x0000ffffffff43c → 0x0000000000000015
$x1 : 0x11
```

Na rysunku 10.15 widać, jak zmieniły się wartości lokalizacji w pamięci tych zmiennych. Zwróć uwagę, że pa i pb nadal zawierają te same adresy, ale wartość zmiennej a się zmieniła, w efekcie czego adres pa wskazuje teraz inną wartość.

```
[SP,#0x20+a]  |: |      0x11        |
[SP,#0x20+b]  |: |      0x11        |
[SP,#0x20+pa] |: | 0000ffffffff43c  | -> [a]: 0x11 (17)
[SP,#0x20+pb] |: | 0000ffffffff438  | -> [b]: 0x11 (17)
[SP,#0x20+t]  |: |      0x15        |
```

Rysunek 10.15. Adresy w pamięci zmiennych a i b teraz zawierają tę samą wartość

Ostatni wiersz funkcji swap zmienia wartość wskazywaną przez pb na wartość zmiennej t, która wynosi 21:

```
Pa = 0x39811ff87c
    *pa = 17
```

Kup książkę Poleć książkę

```
pb = 0x39811ff878
    *pb = 21
t = 21
```

Na rysunku 10.16 widać, że pierwszy rozkaz LDR ładuje zawartość pb do X0. Teraz rejestr X0 zawiera adres wskazujący zawartość zmiennej b:

```
$x0   : 0x0000ffffffff438  →  0x0000001100000011
```

```
void swap(int *pa, int *pb){        swap:
    int t = *pa;                      ...
    *pa = *pb;                        LDR   X0, [SP,#0x20+pb]
    *pb = t;               ——→        LDR   W1, [SP,#0x20+t]
}                                     STR   W1, [X0]
                                      NOP
```

Rysunek 10.16. Ustawienie wartości wskazywanej przez pb na wartość t

Drugi rozkaz LDR ładuje 32 pierwsze bity z lokalizacji na stosie zmiennej t do rejestru W1, po czym rozkaz STR zapisuje wartość W1 pod adresem X0. Po wykonaniu tych trzech rozkazów rejestry X0 i X1 zawierają następujące wartości:

```
$x0   : 0x0000ffffffff438  →  0x0000001100000011
$x1   : 0x15
```

Na rysunku 10.17 widać, jak zmieniła się zawartość lokalizacji zmiennych.

```
[SP,#0x20+a] |: | 0x11             |
[SP,#0x20+b] |: | 0x15             |
[SP,#0x20+pa]|: | 0000ffffffff43c  | -> [a]: 0x11 (17)
[SP,#0x20+pb]|: | 0000ffffffff438  | -> [b]: 0x15 (21)
[SP,#0x20+t] |: | 0x15             |
```

Rysunek 10.17. Zmiany zawartości lokalizacji zmiennych

Program w końcu zwraca wartość do funkcji głównej i otrzymuje instrukcję, aby wydrukować wartości zmiennych a i b. Na rysunku 10.18 widać, że pierwszy rejestr argumentu X0 zostaje ustawiony na lokalizację łańcucha do wydrukowania, a dwa pozostałe rejestry argumentów zawierają wartości przechowywane w lokalizacjach na stosie zmiennych a i b.

Jako że funkcji swap przekazaliśmy kopie adresów zmiennych zamiast ich wartości, funkcja ta operowała na oryginalnych wartościach. Po wywołaniu funkcji printf na ekranie zostaje wydrukowany następujący napis:

```
main: a = 17, b = 21
```

Gdybyśmy zmienne a i b przekazali jako zwykłe wartości całkowitoliczbowe do funkcji swap, która operuje na liczbach całkowitych zamiast na wskaźnikach, to wartości te nie zmieniłyby się w kontekście funkcji głównej.

```
int main(void) {
  ...
  swap(&a, &b);
  printf("main: a = %d, b = %d\n", a, b);
  return 0;
}

  main:
  ...
  BL   swap
  LDR  W0, [SP,#0x20+a]
  LDR  W1, [sp,#0x20+b]
  MOV  W2, W1
  MOV  W1, W0
  ADRL X0, aMainADBD
  BL   .printf
```

Rysunek 10.18. Przygotowanie argumentów do wywołania funkcji printf

Analiza przepływu sterowania

Aby zobaczyć, jak wygląda analiza struktur kontroli przepływu sterowania przez dezasemblację, poddamy inżynierii wstecznej mały program zawierający pętlę `while`, instrukcje `if` i `else` oraz pętlę `for`. Program[7], który będziemy badać, to algorytm konwertujący liczby dziesiętne na szesnastkowe:

```
#include <stdio.h>
void decimal2Hexadecimal(long num);

int main()
{
    long decimalmum;

    printf("Wpisz liczbę dziesiętną: ");
    scanf("%dl", &decimalnum);

    decimal2Hexadecimal(decimalnum);
    return 0;
}
void decimal2Hexadecimal(long num)
{
    long decimalnum = num;
    long quotient, remainder;
    int I, j = 0;
    char hexadecimalnum[100];

    quotient = decimalnum;

    while (quotient != 0)
    {
        remainder = quotient % 16;
```

[7] *github.com/TheAlgorithms/C/blob/2314a195862243e09c485a66194866517a6f8c31/conversions/decimal_to_hexa.c.*

```
        if (remainder < 10)
            hexadecimalnum[j++] = 48 + remainder;
        else
            hexadecimalnum[j++] = 55 + remainder;

        quotient = quotient / 16;
    }

    // Wydruk liczby szesnastkowej

    for (i = j; i >= 0; i- - )
    {
        printf("%c", hexadecimalnum[i]);
    }

    printf("\n");
}
```

Funkcja main

W tym przykładzie wykorzystamy dezasembler programu IDA Pro do analizy statycznej. Jak widać na rysunku 10.19, funkcja `main` drukuje na ekranie łańcuch `Wpisz liczbę dziesiętną` i odbiera od użytkownika dane za pośrednictwem specyfikatora długiej wartości typu `int` (`%ld`) za pomocą funkcji `scanf`. Wartość ta zostaje następnie przekazana do funkcji `decimal2Hexadecimal`.

```
; Attributes: bp-based frame fpd=0x20

; int __cdecl main(int argc, const char **argv, const char **envp)
EXPORT main
main

var_20= -0x20
var_8= -8

; __unwind {
STP             X29, X30, [SP,#var_20]!
MOV             X29, SP
ADRL            X0, aEnterDecimalNu ; "Enter decimal number: "
BL              .printf
ADD             X0, SP, #0x20+var_8
MOV             X1, X0
ADRL            X0, aLd ; "%ld"
BL              .__isoc99_scanf
LDR             X0, [SP,#0x20+var_8]
BL              decimal2Hexadecimal
MOV             W0, #0
LDP             X29, X30, [SP+0x20+var_20],#0x20
RET
; } // starts at 7F4
; End of function main
```

Rysunek 10.19. Dezasemblacja wyniku funkcji main

Zanim program dojdzie do wywołania funkcji `printf`, rozkaz `ADRL` wstawia do rejestru `X0` adres pierwszego łańcucha o etykiecie `aEnterDecimalNu` i przekazuje go do funkcji `printf` jako argument.

Dla funkcji `scanf` program przygotowuje dwa argumenty: adres łańcucha `%ld` w `X0` i lokalizację w pamięci, w której ma zostać zapisana wartość wejściowa. W tym celu rozkaz `ADD` ustawia `X0` na adres `SP` powiększony o offset (`#0x20+var_8`) do lokalizacji, w której ma być przechowywana wartość

wejściowa. A ponieważ powinien to być drugi argument, rozkaz `MOV` kopiuje wartość `X0` do rejestru `X1`, zanim rozkaz `ADRL` wstawi do `X0` adres łańcucha `%ld`.

Po wywołaniu funkcji `scanf` rozkaz `LDR` ładuje wartość (od użytkownika), która była przechowywana na stosie, do `X0`, po czym zostaje ona przekazana jako argument do funkcji `decimal2Hexadecimal`.

Podprocedura

Na początku funkcji `decimal2Hexadecimal` widzimy, że program IDA nadał etykiety różnym offsetom. Służą one do lokalizacji zmiennych funkcji w odniesieniu do wskaźnika stosu (`SP`).

Na rysunku 10.20 widać, że zmieniłam już nazwy domyślnych etykiet zgodnie z nazwami zmiennych funkcji. Zmiana nazw zmiennych podczas analizy to normalna czynność, kiedy już uda się odkryć, do czego służą. Dla zwiększenia czytelności pozostaniemy przy wersji ze zmienionymi nazwami.

Przed

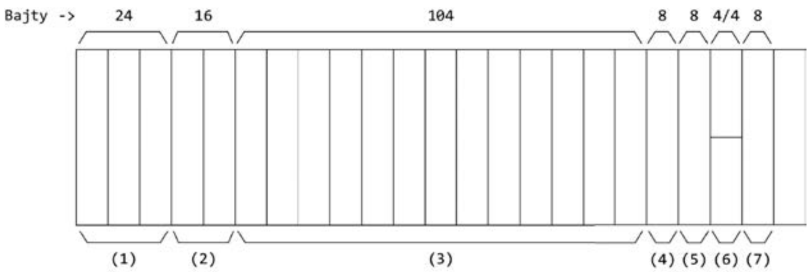
```
; Attributes: bp-based frame fpd=0xB0

; __int64 __fastcall decimal2Hexadecimal(__int64 num)
EXPORT decimal2Hexadecimal
decimal2Hexadecimal

var_B0= -0xB0
var_98= -0x98
var_88= -0x88
var_20= -0x20
var_18= -0x18
var_10= -0x10
var_C= -0xC
var_8= -8
```

Po

```
; Attributes: bp-based frame fpd=0xB0

; __int64 __fastcall decimal2Hexadecimal(__int64 num)
EXPORT decimal2Hexadecimal
decimal2Hexadecimal

var_B0= -0xB0
num= -0x98
hexadecimalnum= -0x88
remainder= -0x20
decimalnum= -0x18
j= -0x10
i= -0xC
quotient= -8
```

Rysunek 10.20. Zmiana nazw lokalnych zmiennych w programie IDA Pro

Jeśli obliczymy odległości między tymi wartościami, to będziemy wiedzieć, jak powinien wyglądać stos oraz gdzie na nim będzie przechowywana każda zmienna. Na rysunku 10.21 pokazałam rozmieszczenie i rozmiary interesujących nas zmiennych.



(1) 24 bajty do zachowania rejestrów (np. X29, X30)
(2) 16 bajtów na liczbę dzisiętną od użytkownika, tylko 8 bajtów zostało użyte (long num)
(3) 104 bajty na tablicę (hexadecimalnum[100])
(4) 8 bajtów na remainder (long)
(5) 8 bajtów na decimalnum (long)
(6) 4 bajty na j (int) i 4 bajty na i (int)
(7) 8 bajtów na quotient (long)

Rysunek 10.21. Układ stosu

Zaczniemy analizę od funkcji decimal2Hexadecimal. Jak widać na rysunku 10.22, pierwszy rozkaz STR zapisuje argument przekazany do tej funkcji przez funkcję wywołującą (main) w specjalnej lokalizacji na stosie. Możemy bezpiecznie założyć, że jest to liczba dziesiętna do konwersji (num). Drugi rozkaz STR zapisuje tę samą wartość w lokalizacji na stosie zmiennej decimalnum, ponieważ te zmienne są inicjalizowane tą samą wartością. Następnie rejestr WZR zostaje użyty do przechowywania 4 bajtów zer na pozycji zmiennej j. Zmienna i zostanie ustawiona później. Na koniec wartość decimalnum zostaje najpierw załadowana do X0, a potem zapisana na pozycji zmiennej quotient, ponieważ quotient = decimalnum.

```
void decimal2Hexadecimal(long num)
{
    long decimalnum = num;
    long quotient, remainder;
    int i, j = 0;
    char hexadecimalnum[100];
    quotient = decimalnum;
...
```

```
decimal2Hexadecimal

STP   X29, X30, [SP,#var_B0]!
MOV   X29, SP
STR   X0, [SP,#0xB0+num]
LDR   X0, [SP,#0xB0+num]
STR   X0, [SP,#0xB0+decimalnum]
STR   WZR, [SP,#0xB0+j]
LDR   X0, [SP,#0xB0+decimalnum]
STR   X0, [SP,#0xB0+quotient]
B     while   ; quotient != 0
```

Rysunek 10.22. Początek funkcji decimal2Hexadecimal

Rozgałęzienie do następnego bloku instrukcji (while) jest bezwarunkowe. Jak pokazałam na rysunku 10.23, ten blok ładuje wartość zmiennej quotient do rejestru X0 i porównuje ją z #0. Rozgałęzienie do bloku instrukcji if_statement zostaje wybrane, kiedy spełniony jest warunek NE (nie równy). Na potrzeby przykładu powiedzmy, że chcemy przekonwertować liczbę dziesiętną 32. Do tej pory zmienne num, decimalnum i quotient były ustawione na 32. To oznacza, że quotient różni się od zera, więc przechodzimy do bloku instrukcji if_statement.

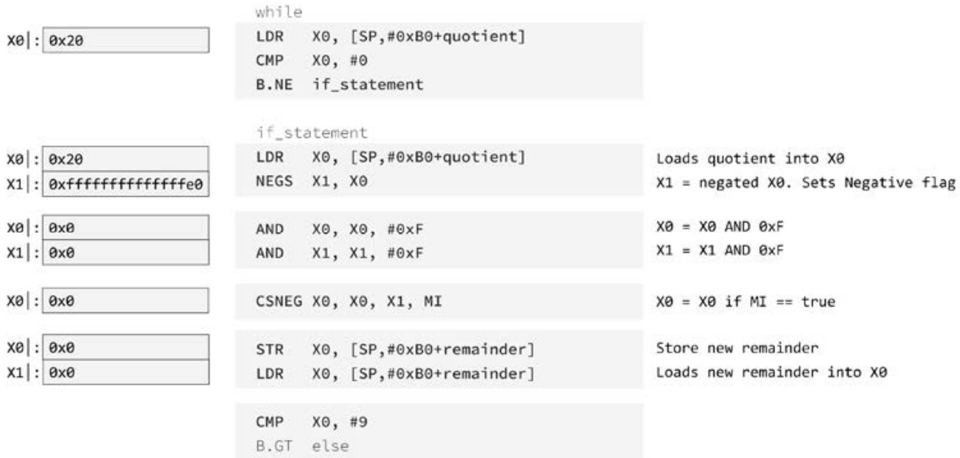
```
Num = 32
Decimalnum = num = 32
Quotient = decimalnum = 32
(quotient != 0) == true
```

```
while (quotient != 0)
{
    remainder = quotient % 16;
    if (remainder < 10)
        hexadecimalnum[j++] = 48 + remainder;

    else
        hexadecimalnum[j++] = 55 + remainder;

    quotient = quotient / 16;
}
```

```
while
LDR    X0, [SP,#0xB0+quotient]
CMP    X0, #0
B.NE   if_statement

if_statement
LDR    X0, [SP,#0xB0+quotient]
NEGS   X1, X0
AND    X0, X0, #0xF
AND    X1, X1, #0xF
CSNEG  X0, X0, X1, MI
STR    X0, [SP,#0xB0+remainder]
LDR    X0, [SP,#0xB0+remainder]
CMP    X0, #9
B.GT   else
```

Rysunek 10.23. Obliczanie wartości zmiennej remainder przy użyciu załadowanej wartości quotient

Blok instrukcji `if_statement` oblicza wartość zmiennej `remainder` poprzez podzielenie modulo wartości zmiennej `quotient` przez 16. Przyjrzymy się każdemu rozkazowi krok po kroku. Na rysunku 10.24 widzimy poszczególne rozkazy oraz użytą lub zmienioną wartość odpowiedniego rejestru.



Rysunek 10.24. Wartości rozkazów i rejestrów

Pierwszy rozkaz `LDR` ładuje obecną wartość `quotient` do `X0`. Mimo że została ona już załadowana do `X0` przez poprzedni rozkaz ładowania, programy często powtarzają tę czynność, na wypadek gdyby wartość `X0` została zmodyfikowana przed dojściem do tego bloku. Rozkaz `NEGS` neguje wartość w `X0`, zapisuje wynik w `X1` i w razie potrzeby ustawia flagę `Negative`. Ten rozkaz jest równoważny z `SUBS X1, XZR, X0`.

Dwa rozkazy `AND` aktualizują rejestry `X0` i `X1` wynikami odpowiednich działań. W naszym przykładzie w obu przypadkach otrzymujemy wynik `0x0`:

```
0x20 & 0xF = 0x0:

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0010 0000
AND
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111
--------------------------------------------------------------------------------
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

0xffffffffffffffe0 & 0xF = 0x0:
1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 0000
AND
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111
--------------------------------------------------------------------------------
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

Rozkaz `CSNEG` (warunkowy wybór i negacja) sprawdza, czy określony warunek (`MI` = ustawiona flaga `Negative`) jest prawdziwy. Jeśli tak, to w docelowym rejestrze (`X0`) zostaje zapisana wartość pierwszego rejestru źródłowego (`X0`). Jeśli warunek jest fałszywy, to w docelowym rejestrze (`X0`) zostaje zapisana zanegowana wartość drugiego rejestru źródłowego (`X1`).

Wynikiem jest nowa reszta (`remainder`). W naszym przykładzie reszta wynosi `0`, ponieważ `32 % 16` = `0`. Rozkaz `STR` zapisuje tę nową resztę na stosie i ładuje tę wartość do `X0`. Na koniec rozkaz `CMP` porównuje wartość w `X0` z `9`. Rozgałęzienie warunkowe sprawdza, czy rozkaz `CMP` ustawił flagę warunkową wymaganą dla warunku `GT`, którym jest > ze znakiem. Jeśli warunek jest prawdziwy, przechodzimy do bloku instrukcji `else`.

Uwaga **Instrukcja `if (remainder < 10)` jest równoznaczna z `if !(remainder > 9)` i nawet jeśli pierwsza wersja zostanie użyta w kodzie źródłowym, kompilator może wyrać inną formę, na przykład taką, jak ta druga.**

W tym momencie pewnie się zastanawiasz, dlaczego nadałam temu blokowi instrukcji nazwę `if_statement`, mimo że większość rozkazów koncentruje się wokół ustawiania wartości zmiennej `remainder` na wynik dzielenia modulo wartości zmiennej `quotient` przez `16`. Wybór nowej nazwy dla bloku instrukcji jest dyktowany tym, jaką logikę chcemy podkreślić i zapamiętać, kiedy wrócimy do tego na późniejszym etapie analizy. W tym przypadku z tego bloku instrukcji przechodzimy do logiki `else` lub do następnej instrukcji w logice instrukcji `if`, zgodnie z rozgałęzieniem warunkowym (`B.GT else`). W widoku grafu programu IDA Pro te dwie części są podzielone na dwa różne bloki instrukcji (zobacz rysunek 10.25). Jednak tylko blok `else` ma etykietę. Jest tak, ponieważ instrukcja znajdująca się za rozgałęzieniem warunkowym jest pierwszą instrukcją lewego bloku i program by po prostu pominął instrukcję rozgałęzienia, kontynuując działanie sekwencyjnie, gdyby warunek nie został spełniony.

```
while (quotient != 0)
{
    remainder = quotient % 16;
    if (remainder < 10)
        hexadecimalnum[j++] = 48 + remainder;

    else
        hexadecimalnum[j++] = 55 + remainder;

    quotient = quotient / 16;
}
```

```
if_statement
LDR   X0, [SP,#0xB0+quotient]
NEGS  X1, X0
AND   X0, X0, #0xF
AND   X1, X1, #0xF
CSNEG X0, X0, X1, MI
STR   X0, [SP,#0xB0+remainder]
LDR   X0, [SP,#0xB0+remainder]
CMP   X0, #9
B.GT  else
```

X0 > 9 -> False

```
LDR   X0, [SP,#0xB0+remainder]
AND   W1, W0, #0xFF
LDR   W0, [SP,#0xB0+j]
ADD   W2, W0, #1
STR   W2, [SP,#0xB0+j]
ADD   W1, W1, #0x30
AND   W2, W1, #0xFF
SXTW  X0, W0
ADD   X1, SP, #0xB0+hexadecimalnum
STRB  W2, [X1,X0]
B     divide_quotient
```

X0 > 9 -> True

```
else
LDR   X0, [SP,#0xB0+remainder]
AND   W1, W0, #0xFF
LDR   W0, [SP,#0xB0+j]
ADD   W2, W0, #1
STR   W2, [SP,#0xB0+j]
ADD   W1, W1, #0x37
AND   W2, W1, #0xFF
SXTW  X0, W0
ADD   X1, SP, #0xB0+hexadecimalnum
STRB  W2, [X1,X0]
```

Rysunek 10.25. Rozgałęzienie warunkowe oparte na instrukcji if-else

Kup ksi k Pole ksi k

Wracając do naszego przykładu, obecnie zmienna remainder ma wartość 0. Rozkaz CMP wewnętrznie oblicza 0 – 9 = –9 i ustawia tylko flagę N (Negative). W efekcie warunek GT jest fałszywy, dlatego pomijamy rozgałęzienie do bloku else. Innymi słowy, 0 nie jest większe niż 9, więc przechodzimy do następnej instrukcji.

Konwersja na char

Zanim wejdziemy do bloków instrukcji if i else, przypomnimy sobie pokrótce, co właściwie tu się dzieje. Może zastanawiasz się, dlaczego algorytm dodaje 48 do reszty, jeśli jest ona mniejsza od 10, i 55, jeśli jest większa. Aby to zrozumieć, musisz sobie przypomnieć dwie rzeczy: wstawiamy wartości do tablicy typu char i oczekujemy wyniku w postaci sekwencji wartości typu char.

Powiedzmy, że chcemy przekonwertować wartość dziesiętną 171 na szesnastkową. Obliczenia wyglądają następująco:

171 : 16 = 10 (reszta 11)

10 : 16 = 0 (reszta 10)

Bierzemy reszty, zaczynając od ostatniej, i konwertujemy je na format szesnastkowy:

10 = 0xA

11 = 0xB

Wynik: AB

To jest oczywiste dla nas, ale nie dla komputera. Ten polega na formacie wyjściowym, którym w tym przypadku jest sekwencja wartości typu char z tablicy typu char. Jeśli tablica zawiera wartości 10 (0xA) i 11 (0xB) i otrzyma instrukcję zwrócenia ich ekwiwalentów w formacie char, to dostaniemy znak nowego wiersza i tabulator pionowy. Przyczyną tego jest to, że zgodnie z tabelą znaków ASCII te wartości reprezentują znaki \n i \v (zobacz tabelę 10.1).

Tabela 10.1. Tabela ASCII

Forma dziesiętna	Forma szesnastkowa	Znak
10	0A	\n (nowy wiersz)
11	0B	\v (tabulator pionowy)

Jak tego uniknąć? Spoglądając na tabelę znaków ASCII, dowiemy się, że aby zwrócić znaki A i B, tablica musi zawierać wartości odpowiadające tym znakom. Oznacza to, że musimy dodać 55 (0x37) do obu wartości, aby otrzymać 65 (0x41) w przypadku wartości 10 oraz 66 (0x42) w przypadku wartości 11 (zobacz tabelę 10.2).

Tabela 10.2. Tabela ASCII

Reszta (dziesiętna)	Reszta + 55 (dziesiętna)	Forma szesnastkowa	Znak
10	65	0x41	A
11	66	0x42	B

To daje nam odpowiedź na pytanie, dlaczego dodajemy 55 do reszty, ale wciąż nie wiemy, dlaczego dodajemy 48, jeśli reszta jest mniejsza od 10. Kiedy już się dowiesz, powód stanie się prosty.

Powiedzmy, że chcemy przekonwertować dziesiętną wartość 32 na format szesnastkowy. Bierzemy reszty, zaczynając od ostatniej, i konwertujemy je na wartości szesnastkowe:

32 : 16 = 2 (reszta 0)

2 : 16 = 0 (reszta 2)

2 = `0x2`

0 = `0x0`

Wynik: `0x20`

W tabeli ASCII widzimy, że numeryczne znaki od ich dziesiętnych i szesnastkowych ekwiwalentów dzieli 48, a nie 55 pozycji (zobacz tabelę 10.3).

Tabela 10.3. Tabela ASCII

Reszta (dziesiętna)	Reszta + 55 (dziesiętna)	Forma szesnastkowa	Znak
0	48	0x30	0
2	50	0x32	2

Gdyby jednak reszta była większa od 9, skończylibyśmy ze znakami specjalnymi, takimi jak < lub = (zobacz tabelę 10.4). Na przykład jeśli reszta wynosi 10 i dodamy do niej 48, to otrzymamy 58 w formacie dziesiętnym lub `0x3A` w formacie szesnastkowym, czyli wartość reprezentującą dwukropek w `char`. Dlatego pomijamy te znaki przez dodanie 55 (`0x37`) zamiast 48 (`0x30`), jeśli reszta jest większa od 9. Zawsze pozostajemy w przedziale `0-9` i `A-F`, ponieważ reszta jest obliczana przez dzielenie modulo wartości zmiennej `quotient` przez 16 (`remainder = quotient % 16`).

Tabela 10.4. Tabela ASCII

Format dziesiętny	Format szesnastkowy	Znak
55	**37**	**7**
56	**38**	**8**
57	**39**	**9**
58	3A	:
59	3B	;
60	3C	<
61	3D	=

Instrukcja if

Przechodzimy do logiki wewnątrz instrukcji `if`. Spójrz na rysunek 10.26.

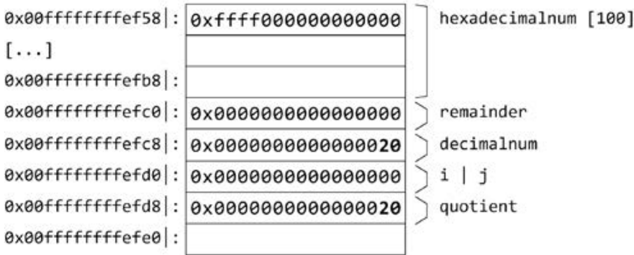
Kup ksi k Pole ksi k

```
while (quotient != 0)
{
    remainder = quotient % 16;
    if (remainder < 10)
        hexadecimalnum[j++] = 48 + remainder;

    else
        hexadecimalnum[j++] = 55 + remainder;

    quotient = quotient / 16;
}
```

```
B.GT  else

LDR   X0, [SP,#0xB0+remainder]
AND   W1, W0, #0xFF
LDR   W0, [SP,#0xB0+j]
ADD   W2, W0, #1
STR   W2, [SP,#0xB0+j]
ADD   W1, W1, #0x30
AND   W2, W1, #0xFF
SXTW  X0, W0
ADD   X1, SP, #0xB0+hexadecimalnum
STRB  W2, [X1,X0]
B     divide_quotient
```

Rysunek 10.26. Instrukcja if

W tym punkcie naszych obliczeń bieżąca reszta wynosi `0` (`remainder = 32 % 16`), a stos wygląda tak, jak na rysunku 10.27.



Rysunek 10.27. Stos

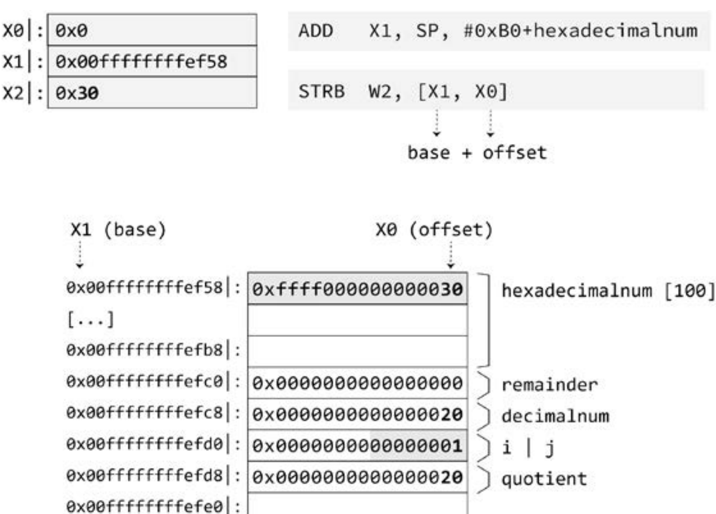
Na rysunku 10.28 widzimy, że pierwszy rozkaz ładuje resztę do `X0`, po czym rozkaz `AND` ustawia `W1` na `remainder & 255`, aby wartość pozostała w zakresie jednego bajta. Następne trzy rozkazy ładują bieżącą wartość `j` (`0`) ze stosu, zwiększają ją o 1 i zapisują wynik do użycia w następnej iteracji.

```
X0|: 0x0      LDR   X0, [SP,#0xB0+remainder]   remainder = 0
X1|: 0x0      AND   W1, W0, #0xFF              remainder AND 0xFF = 0

X0|: 0x0      LDR   W0, [SP,#0xB0+j]           j = 0
X2|: 0x1      ADD   W2, W0, #1                 j = j + 1
              STR   W2, [SP,#0xB0+j]           store j = 1 on stack

X1|: 0x30     ADD   W1, W1, #0x30              remainder + 0x30 (48) = 0x30
X2|: 0x30     AND   W2, W1, #0xFF              result AND 0xFF (255) = 0x30

X0|: 0x0      SXTW  X0, W0                     Sign-extend W0 to X0 = 0x0
```

Rysunek 10.28. Instrukcja if po dezasemblacji

Potem rozkaz `ADD` dodaje `0x30 (48)` do wartości reszty w `W1`. Operacja `AND` pobiera nową wartość reszty z `W1`, stosuje `AND 0xFF (255)`, aby wartość pozostała w granicach jednego bajta, i wstawia wynik do `W2`. Rozkaz `SXTW` rozszerza znakiem 32 pierwsze bity (`W0`) do 64 bitów (`X0`), aby druga połowa 64-bitowego rejestru `X0` była wyzerowana, ponieważ poprzednio zostały zmodyfikowane tylko 32 bity (`W0`).

Czas zapisać pierwszy wynik w tablicy. Jak pokazałam na rysunku 10.29, rozkaz `ADD` umieszcza w rejestrze `X1` adres tablicy `hexadecimalnum`. Rozkaz `STRB` zapisuje wartość `W2` pod adresem znajdującym się w `X1` (baza: adres stosu tablicy) + `X0` (offset: `j`). Zwróć uwagę, że choć wartość `j` zostaje zwiększona o 1 i zaktualizowana na swojej pozycji na stosie, wartość zostaje zapisana w elemencie `j = 0`.



Rysunek 10.29. Zapisywanie pierwszego wyniku w tablicy

Dzielenie współczynnika

Następnie program dochodzi do wiersza ustawiającego nowy współczynnik (`quotient`) przez podzielenie wartości zmiennej `quotient` przez 16 (zobacz rysunek 10.30). W naszym przykładzie obecna wartość współczynnika nadal wynosi 32.

```
while (quotient != 0)
{
    remainder = quotient % 16;
    if (remainder < 10)
        hexadecimalnum[j++] = 48 + remainder;

    else
        hexadecimalnum[j++] = 55 + remainder;

    quotient = quotient / 16;
}
```

```
divide_quotient

LDR   X0, [SP,#0xB0+quotient]
ADD   X1, X0, #0xF
CMP   X0, #0
CSEL  X0, X1, X0, LT
ASR   X0, X0, #4
STR   X0, [SP,#0xB0+quotient]
```

Rysunek 10.30. Dzielenie współczynnika

W ramach przygotowania do tego dzielenia rozkaz `ADD` dodaje 15 do wartości współczynnika (`X0`) i wstawia wynik do `X1` (zobacz rysunek 10.31). Rozkaz `CMP` porównuje wartość współczynnika z `#0` oraz ustawia flagi warunkowe w ramach przygotowania do rozkazu `CSEL`.

Kup ksi k Pole ksi k

```
divide_quotient

X0|: 0x20      LDR   X0, [SP,#0xB0+quotient]
X1|: 0x2F      ADD   X1, X0, #0xF              ; X1 = X0 (quotient) + 15

               CMP   X0, #0                    ; X0 - #0 -> set flags
               CSEL  X0, X1, X0, LT            ; LT == False: X0 = X0

X0|: 0x2       ASR   X0, X0, #4                ; quotient = quotient / 2
               STR   X0, [SP,#0xB0+quotient]   ; store new quotient
```

Rysunek 10.31. Dzielenie współczynnika, analiza kodu po dezasemblacji

Rozkaz `CSEL` (wybór warunkowy) sprawdza, czy określony warunek (`LT`) jest prawdziwy, a jeśli tak, zapisuje wartość pierwszego rejestru źródłowego (`X1`) w rejestrze docelowym (`X0`). Jeśli warunek jest fałszywy, co oznacza, że współczynnik nie jest ujemny, zapisuje wartość drugiego rejestru źródłowego (`X0`) do rejestru docelowego (`X0`). W naszym przypadku poprzedni rozkaz `CMP` nie ustawił flagi ujemności `N`, a więc warunek jest fałszywy. To oznacza, że wartość w `X0` pozostanie niezmieniona.

Rozkaz `ASR` stosuje przesunięcie arytmetyczne w prawo o 4 do wartości w `X0` (współczynnik), co w efekcie oznacza podzielenie jej przez 16. Rozkaz `STR` zapisuje nowy współczynnik w odpowiednim miejscu na stosie.

Po podzieleniu współczynnika wracamy do bloku instrukcji sprawdzającego warunek pętli `while` (zobacz rysunek 10.32). Ponieważ obecny współczynnik ma wartość `0x2`, powtarzamy obliczenia reszty i dochodzimy do instrukcji `if`, która dodaje do reszty `48`, po czym następuje dzielenie współczynnika w sposób już opisany. W naszym przykładzie blok `else` nigdy nie zostaje wykonany, gdyż reszta zawsze jest mniejsza niż 10.

```
while (quotient != 0)
{
    remainder = quotient % 16;
    if (remainder < 10)
        hexadecimalnum[j++] = 48 + remainder;

    else
        hexadecimalnum[j++] = 55 + remainder;

    quotient = quotient / 16;
}
```

```
while
LDR   X0, [SP,#0xB0+quotient]
CMP   X0, #0
B.NE  if_statement

[...]

divide_quotient
LDR   X0, [SP,#0xB0+quotient]
ADD   X1, X0, #0xF
CMP   X0, #0
CSEL  X0, X1, X0, LT
ASR   X0, X0, #4
STR   X0, [SP,#0xB0+quotient]
```

Rysunek 10.32. Sprawdzenie warunku pętli while

Pętla for

Po drugiej iteracji wartość współczynnika wynosi 0x0, a reszta — 0x2. Oznacza to, że wychodzimy z pętli while i wchodzimy do pętli for. Na rysunku 10.33 jest pokazany układ stosu w tym momencie. W tablicy hexadecimalnum znajdują się wartości 0x00, 0x30 oraz 0x32, a zmienna j ma wartość 0x2.

```
0x00ffffffffef48|: 0x0000000000000020   num
0x00ffffffffef50|:
0x00ffffffffef58|: 0xffff000000003230   hexadecimalnum [100]
[...]
0x00ffffffffefb8|:
0x00ffffffffefc0|: 0x0000000000000002   remainder
0x00ffffffffefc8|: 0x0000000000000020   decimalnum
0x00ffffffffefd0|: 0x0000000000000002   i | j
0x00ffffffffefd8|: 0x0000000000000000   quotient
0x00ffffffffefe0|:
```

Rysunek 10.33. Obecny układ stosu

Pętla for ustawia wartość i na bieżącą wartość j i sprawdza, czy wartość i jest większa od 0 lub mu równa (zobacz rysunek 10.34).

```
for (i = j; i >= 0; i--)
{
    printf("%c", hexadecimalnum[i]);
}
```

```
while
LDR   X0, [SP,#0xB0+quotient]
CMP   X0, #0
B.NE  if_statement              ; NE => False

LDR   W0, [SP,#0xB0+j]          ; Load j
STR   W0, [SP,#0xB0+i]          ; Store i = j
B     loc_918

loc_918
LDR   W0, [SP,#0xB0+i]          ; Load i
CMP   W0, #0                    ; i >= 0?
B.GE  loc_8FC                   ; branch if true
```

Rysunek 10.34. Pętla for

Aby ustawić zmienną i na wartość j, wystarczy zapisać tę samą wartość w lokalizacji na stosie zmiennej i. Widzimy, że rozkaz LDR ładuje wartość j do rejestru W0, a rozkaz STR zapisuje tę wartość na pozycji i, która znajduje się o cztery bajty niżej od pozycji j. Potem następuje bezwarunkowe rozgałęzienie do następnego bloku instrukcji.

W tym momencie zostaje sprawdzony warunek pętli for. Obecna wartość zmiennej i po raz kolejny zostaje załadowana do W0, po czym rozkaz CMP porównuje wartość w W0 (i) z #0 i odpowiednio ustawia flagi warunkowe. Jeśli flagi te wskazują, że wartość i jest większa (GE) niż #0, przechodzimy do bloku instrukcji loc_8FC. W naszym przykładzie bieżąca wartość i wynosi 2, a więc warunek GE (N==V) jest prawdziwy, w związku z czym rozgałęzienie zostaje wybrane.

Dochodzimy do bloku instrukcji odpowiedzialnego za wydrukowanie znaku tablicy hexadecimalnum na pozycji i oraz zmniejszenie wartości i (zobacz rysunek 10.35).

```
for (i = j; i >= 0; i--)
{
    printf("%c", hexadecimalnum[i]);
}
```

```
loc_918
LDR    W0, [SP,#0xB0+i]
CMP    W0, #0
B.GE   loc_8FC

loc_8FC
LDRSW  X0, [SP,#0xB0+i]
ADD    X1, SP, #0xB0+hexadecimalnum
LDRB   W0, [X1,X0]
BL     .putchar
LDR    W0, [SP,#0xB0+i]
SUB    W0, W0, #1
STR    W0, [SP,#0xB0+i]
```

Rysunek 10.35. Drukowanie znaku z tablicy hexadecimalnum

Przyjrzyjmy się tym rozkazom po kolei. Na rysunku 10.36 widzimy, że pierwszym rozkazem jest `LDRSW`, który ładuje słowo ze znakiem (32 bity) do rejestru docelowego `X0`.

```
                                loc_8FC
X0|: 0x2                  LDRSW  X0, [SP,#0xB0+i]              ; Załadowanie i do X0
X1|: 0x00ffffffffef58     ADD    X1, SP, #0xB0+hexadecimalnum  ; X1 to adres tablicy

X0|: 0x00                 LDRB   W0, [X1,X0]                   ; Załadowanie bajta spod adresu X1+X0
                          BL     .putchar                      ; Wydrukowanie znaku

X0|: 0x2                  LDR    W0, [SP,#0xB0+i]              ; Załadowanie i do W0
X0|: 0x1                  SUB    W0, W0, #1                    ; i = i - 1
                          STR    W0, [SP,#0xB0+i]              ; Zapisanie nowej wartości i
```

Rysunek 10.36. Analiza kodu po dezasemblacji

To jest obecna wartość `i` (`0x2`), która służy jako offset do tablicy. Rozkaz `ADD` wstawia do rejestru `X1` adres tablicy `hexadecimalnum`. Wartość ta służy jako adres bazowy. Aby funkcja `putchar` wydrukowała znak, musi otrzymać wartość typu `char` przekazaną jako argument (`X0`/`W0`). Zatem rozkaz `LDRB` ładuje bajt z adresu bazowego (`X1`) przesuniętego o offset `i` (`X0`) do rejestru docelowego `W0`. Jeśli nie możesz połapać się w rozkładzie elementów na stosie, spójrz na rysunek 10.37. Adres bazowy wskazuje pierwszy element tablicy, zawierający wartość `0x30`. Jako że obecnie zmienna `i` ma wartość `2`, rozkaz `LDRB` pobiera `hexadecimalnum[2] = 0x00`.

```
[7][6][5][4][3][2][1][0]
 FF FF 00 00 00 00 32 30 : |0x00ffffffffef58

                  ... [9][8]
 00 00 00 00 00 00 00 00 : |0x00ffffffffef60
```

Rysunek 10.37. Rozmieszczenie elementów

Po wywołaniu funkcji `putchar` wartość `i` musi zostać zmniejszona. Rozkaz `LDR` ładuje wartość do `W0`, odejmuje od niej `1` za pomocą rozkazu `SUB` i zapisuje nową wartość na stosie.

Ta pętla kontynuuje działanie, dopóki wartość `i` jest mniejsza od `0` i warunek `GE` instrukcji rozgałęzienia nie jest spełniony. W tym przypadku dochodzimy do wiersza `printf("\n")` (zobacz rysunek 10.38). Teraz rozkaz `MOV` ustawia pierwszy argument funkcji `putchar` na szesnastkowy odpowiednik znaku nowego wiersza (`\n`), po czym następuje rozgałęzienie do `putchar`.

```
[...]
    for (i = j; i >= 0; i--)
    {
        printf("%c", hexadecimalnum[i]);
    }

    printf("\n");
}
```

```
loc_918
LDR    W0, [SP,#0xB0+i]
CMP    W0, #0
B.GE   loc_8FC                  ; GE => fałsz

MOV    W0, #0xA                 ; W0 = 0xA (\n)
BL     .putchar                 ; Wydrukowanie znaku
NOP
LDP    X29, X30, [SP], #176     ; Przywrócenie X29 i X30
RET                             ; Powrót
```

Rysunek 10.38. Drukowanie wiersza

Analiza algorytmu

W poprzednim podrozdziale przyjrzeliśmy się działaniu wskaźników w kodzie asemblera, przeanalizowaliśmy przepływ sterowania programu oraz porównaliśmy fragmenty kodu źródłowego z kodem dezasemblera. W tym rozdziale przeanalizujemy nieznany algorytm bez patrzenia na pierwotny kod źródłowy i na pseudokod z dekompilatora. Naszym celem jest przećwiczenie analizy przepływu warunkowego w zdezasemblowanym kodzie i odszyfrowanie znaczenia każdego rozkazu.

Oto zdezasemblowany kod funkcji `main` i `algoFunc` uzyskany za pomocą narzędzia `objdump`:

```
0000000000000918 <main>:
 918:   a9be7bfd     stp    x29, x30, [sp, #-32]!
 91c:   910003fd     mov    x29, sp
 920:   90000000     adrp   x0, 0 <_init-0x6a8>
 924:   9128a000     add    x0, x0, #0xa28
 928:   97ffff8a     bl     750 <printf@plt>
 92c:   910063e0     add    x0, sp, #0x18
 930:   aa0003e1     mov    x1, x0
 934:   90000000     adrp   x0, 0 <_init-0x6a8>
 938:   9128e000     add    x0, x0, #0xa38
 93c:   97ffff81     bl     740 <__isoc99_scanf@plt>
 940:   b9401be0     ldr    w0, [sp, #24]
 944:   97ffffc8     bl     864 <algoFunc>
 948:   39007fe0     strb   w0, [sp, #31]
 94c:   39407fe0     ldrb   w0, [sp, #31]
 950:   7100001f     cmp    w0, #0x0
 954:   540000a0     b.eq   968 <main+0x50>             // b.none
 958:   90000000     adrp   x0, 0 <_init-0x6a8>
 95c:   91290000     add    x0, x0, #0xa40
 960:   97ffff74     bl     730 <puts@plt>
 964:   14000006     b      97c <main+0x64>
 968:   b9401be0     ldr    w0, [sp, #24]
 96c:   2a0003e1     mov    w1, w0
 970:   90000000     adrp   x0, 0 <_init-0x6a8>
 974:   91296000     add    x0, x0, #0xa58
 978:   97ffff76     bl     750 <printf@plt>
```

```
 97c:   52800000    mov    w0, #0x0                    // #0
 980:   a8c27bfd    ldp    x29, x30, [sp], #32
 984:   d65f03c0    ret

0000000000000864 <algoFunc>:
 864:   a9bd7bfd    stp    x29, x30, [sp, #-48]!
 868:   910003fd    mov    x29, sp
 86c:   b9001fe0    str    w0, [sp, #28]
 870:   b9401fe0    ldr    w0, [sp, #28]
 874:   7100081f    cmp    w0, #0x2
 878:   54000061    b.ne   884 <algoFunc+0x20>         // b.any
 87c:   52800020    mov    w0, #0x1                    // #1
 880:   14000024    b      910 <algoFunc+0xac>
 884:   b9401fe0    ldr    w0, [sp, #28]
 888:   7100041f    cmp    w0, #0x1
 88c:   540000ad    b.le   8a0 <algoFunc+0x3c>
 890:   b9401fe0    ldr    w0, [sp, #28]
 894:   12000000    and    w0, w0, #0x1
 898:   7100001f    cmp    w0, #0x0
 89c:   54000061    b.ne   8a8 <algoFunc+0x44>         // b.any
 8a0:   52800000    mov    w0, #0x0                    // #0
 8a4:   1400001b    b      910 <algoFunc+0xac>
 8a8:   b9401fe0    ldr    w0, [sp, #28]
 8ac:   1e620000    scvtf  d0, w0
 8b0:   97ffff90    bl     6f0 <sqrt@plt>
 8b4:   fd0013e0    str    d0, [sp, #32]
 8b8:   52800060    mov    w0, #0x3                    // #3
 8bc:   b9002fe0    str    w0, [sp, #44]
 8c0:   1400000e    b      8f8 <algoFunc+0x94>
 8c4:   b9401fe0    ldr    w0, [sp, #28]
 8c8:   b9402fe1    ldr    w1, [sp, #44]
 8cc:   1ac10c02    sdiv   w2, w0, w1
 8d0:   b9402fe1    ldr    w1, [sp, #44]
 8d4:   1b017c41    mul    w1, w2, w1
 8d8:   4b010000    sub    w0, w0, w1
 8dc:   7100001f    cmp    w0, #0x0
 8e0:   54000061    b.ne   8ec <algoFunc+0x88>         // b.any
 8e4:   52800000    mov    w0, #0x0                    // #0
 8e8:   1400000a    b      910 <algoFunc+0xac>
 8ec:   b9402fe0    ldr    w0, [sp, #44]
 8f0:   11000800    add    w0, w0, #0x2
 8f4:   b9002fe0    str    w0, [sp, #44]
 8f8:   b9402fe0    ldr    w0, [sp, #44]
 8fc:   1e620000    scvtf  d0, w0
 900:   fd4013e1    ldr    d1, [sp, #32]
 904:   1e602030    fcmpe  d1, d0
 908:   54fffdea    b.ge   8c4 <algoFunc+0x60>         // b.tcont
 90c:   52800020    mov    w0, #0x1                    // #1
 910:   a8c37bfd    ldp    x29, x30, [sp], #48
 914:   d65f03c0    ret
```

Zanim zagłębimy się w algorytm `algoFunc`, musimy zidentyfikować argumenty, które przekazuje do niego funkcja wywołująca (w tym przypadku `main`).

Na rysunku 10.39 widzimy funkcję `main` i trzy zmienne lokalne w ramce stosu. Mają one nazwy `var_x`, gdzie x jest szesnastkową wartością offsetu lokalizacji w ramce stosu.

```
; Attributes: bp-based frame fpd=0x20

; int __cdecl main(int argc, const char **argv, const char **envp)
EXPORT main
main

var_20= -0x20
var_8= -8
var_1= -1

; __unwind {
STP             X29, X30, [SP,#var_20]!
MOV             X29, SP
ADRL            X0, aPickANumber ; "Pick a number: "
BL              .printf
ADD             X0, SP, #0x20+var_8
MOV             X1, X0
ADRL            X0, aD  ; "%d"
BL              .__isoc99_scanf
LDR             W0, [SP,#0x20+var_8]
BL              algoFunc
STRB            W0, [SP,#0x20+var_1]
LDRB            W0, [SP,#0x20+var_1]
CMP             W0, #0
B.EQ            loc_968
```

Rysunek 10.39. Widok zdezasemblowanej funkcji main

Pierwsze wywołanie funkcji dotyczy funkcji `printf`, która drukuje łańcuch `"Pick a number"` (wybierz liczbę). Ma on etykietę `aPickANumber` przypisaną do jego lokalizacji, która jest załadowana do `X0` przez rozkaz `ADRL` i służy jako argument wywołania `printf`.

Funkcja `scanf` przyjmuje dwa argumenty, które są zapisane w rejestrach `X0` i `X1`. Po wykonaniu trzech rozkazów znajdujących się za wywołaniem funkcji `printf` pierwszy argument (`X0`) wskazuje deskryptor formatu `%d`, a drugi argument (`X1`) zawiera adres na stosie, w którym zostaną zapisane dane od użytkownika.

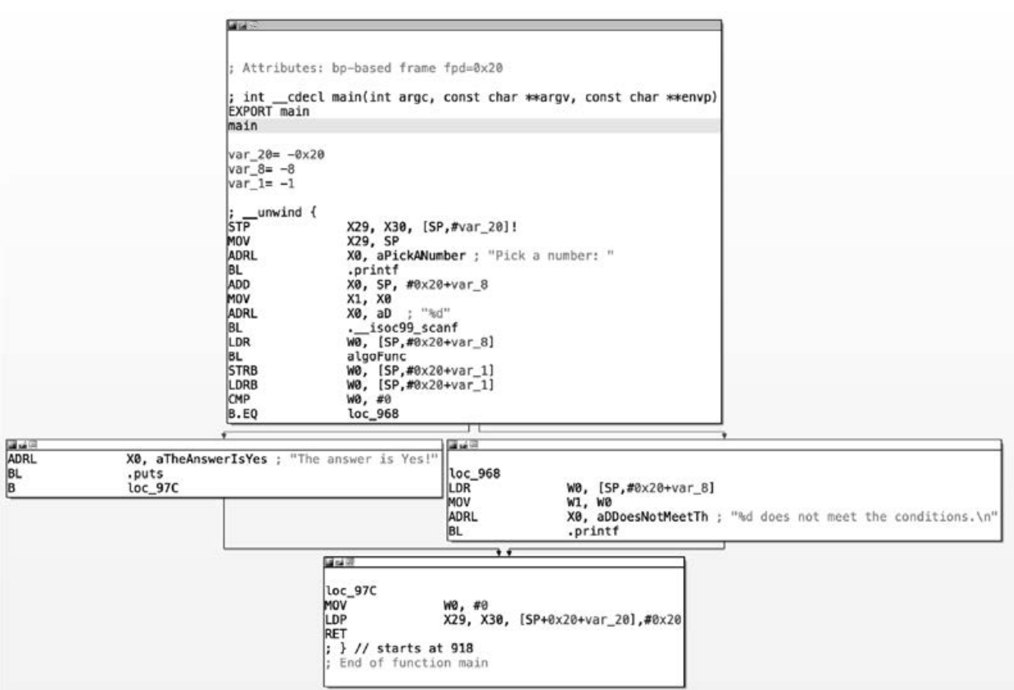
Po wykonaniu funkcji `scanf` program ładuje dane od użytkownika z lokalizacji `[SP,#0x20+var_8]` na stosie do rejestru `W0`. Funkcja `algoFunc` przyjmuje tylko jeden argument, którym są dane wprowadzone przez użytkownika przechowywane w `W0`.

Kiedy funkcja `algoFunc` zwraca wartość, jeden bajt tej wartości zwrotnej zostaje zapisany na stosie i porównany z liczbą `0`. Jeśli wartością zwrotną jest `0`, program dokonuje rozgałęzienia do bloku instrukcji, który drukuje łańcuch informujący, że liczba nie spełnia warunków (spójrz na prawy blok instrukcji na rysunku 10.40). Jeśli wartością zwrotną jest `1`, program dokonuje rozgałęzienia do bloku instrukcji informującego, że odpowiedź jest twierdząca.

Chcemy odtworzyć algorytm funkcji `algoFunc` i dowiedzieć się, jakich liczb ona oczekuje, aby wydrukować informację, że odpowiedź jest poprawna. Na rysunku 10.41 jest przedstawiony schemat przepływu sterowania funkcji `algoFunc`.

Rozpocznijmy analizę tej logiki. Na rysunku 10.42 widzimy, że program IDA przypisał etykiety czterem zmiennym lokalnym, które są używane w tej funkcji. Pierwsza z nich, `var_30`, to miejsce przechowywania pierwotnych wartości `X32` i `X30` na stosie za pomocą rozkazu `STP`. Lokalizacja wartości wprowadzonej przez użytkownika do tej funkcji przez `W0` jest oznaczona etykietą `var_14`.

Kup ksi k Pole ksi k



Rysunek 10.40. Rozgałęzienie warunkowe na podstawie wartości zwrotnej funkcji algoFunc

W tym momencie nie mamy informacji pozwalających stwierdzić, do czego służą zmienne `var_10` i `var_4`. Kiedy wartość wprowadzona przez użytkownika zostanie zapisana na stosie, rozkaz `CMP` porównuje ją z liczbą `2`, po czym następuje rozgałęzienie do bloku instrukcji `loc_884`, jeśli wartości nie są równe (`NE`). Jeżeli liczba podana przez użytkownika jest równa `2`, rejestr `W0` zostaje ustawiony na `1` i program przechodzi do bloku instrukcji `loc_910`, którego zadaniem jest powrót do funkcji głównej i przekazanie wartości zwrotnej przez `W0`.

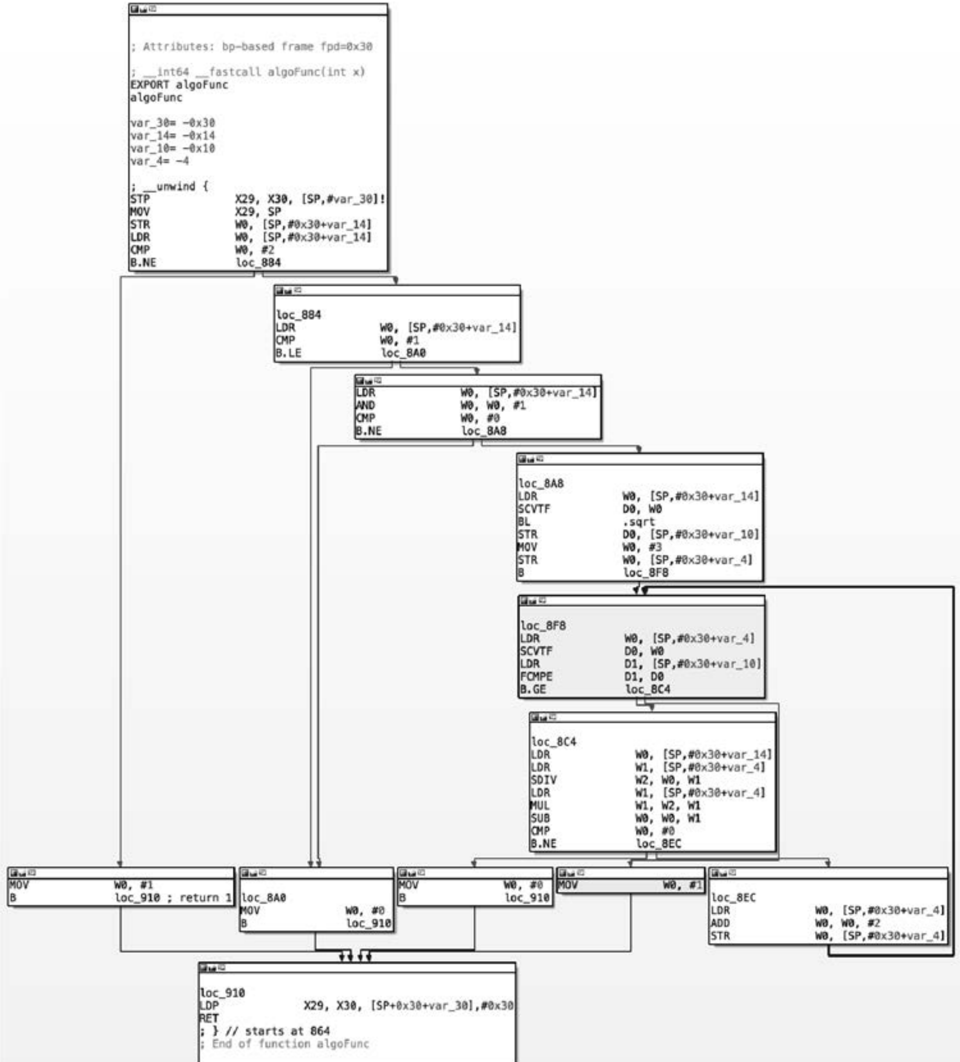
Na razie mamy następujące informacje:

- Zmienna `var_14` odpowiada wartości wpisanej przez użytkownika.
- Zmienna `var_30` to miejsce przechowywania wartości rejestrów `X29` i `X30` w ramce stosu.
- Podprocedura zwraca wartość `1`, jeśli użytkownik wpisze wartość `2`.

Oznacza to, że pierwszy fragment pseudokodu wygląda tak:

```
if ( x == 2 )
    return 1;
```

Po zapisaniu wartości wprowadzonej przez użytkownika na stosie rozkaz `CMP` porównuje ją z `2`, po czym następuje wykonanie rozkazu `B.NE`, który dokonuje rozgałęzienia do bloku instrukcji `loc_884`, jeśli warunek `NE` (nie równy) jest spełniony. Innymi słowy, jeśli wartość w `W0` jest inna niż `2`, program przechodzi do bloku instrukcji po prawej.



Rysunek 10.41. Schemat przepływu sterowania funkcji algoFunc

```
algoFunc                                ; CODE XREF: main+2C↓p

var_30          = -0x30
var_14          = -0x14
var_10          = -0x10
var_4           = -4

; __unwind {
                STP             X29, X30, [SP,#var_30]!
                MOV             X29, SP
                STR             W0, [SP,#0x30+var_14]
                LDR             W0, [SP,#0x30+var_14]
                CMP             W0, #2
                B.NE            loc_884
                MOV             W0, #1
                B               loc_910 ; return 1
```

Rysunek 10.42. Etykiety zmiennych lokalnych przypisane przez program IDA Pro

Wybierzemy losową liczbę, aby pomóc sobie w obliczeniach: 14. W tym przypadku program przechodzi do bloku loc_884, ponieważ liczby 14 i 2 nie są równe.

Wartość wpisana przez użytkownika zostaje załadowana do W0 i porównana z 1. Jeśli wartość w W0 jest mniejsza lub równa (LE) 1, program przechodzi do gałęzi loc_8A0. Jak widać na rysunku 10.43, to nie jest blok instrukcji, do którego chcemy trafić, ponieważ ustawia on W0 na 0 i przechodzi do bloku loc_910, który wraca do funkcji głównej. Pamiętaj, że jeśli funkcja algoFunc zwraca 0, oznacza to, że wprowadziliśmy niepoprawne dane.

```
loc_884                         ; CODE XREF: algoFunc+14↑j
            LDR         W0, [SP,#0x30+input]
            CMP         W0, #1
            B.LE        loc_8A0
            LDR         W0, [SP,#0x30+input]
            AND         W0, W0, #1
            CMP         W0, #0
            B.NE        loc_8A8

loc_8A0                         ; CODE XREF: algoFunc+28↑j
            MOV         W0, #0
            B           loc_910
```

Rysunek 10.43. Blok loc_8A0

W naszym przypadku liczba 14 nie jest mniejsza od 1 ani jej równa, więc wykonujemy rozkaz LDR po rozgałęzieniu. Zanim dojdziemy do następnej instrukcji rozgałęzienia, do wartości wejściowej zostaje zastosowana operacja AND 1, po czym następuje porównanie z 0.

Jeśli zastosujemy to do liczby 14, wynikiem operacji AND będzie 0. Oznacza to, że gałąź B.NE nie zostanie wykonana i przejdziemy do bloku instrukcji loc_910, który zwraca do funkcji wywołującej main wartość 0. Wiemy już, że liczba 14 nie spełnia kryteriów algorytmu. Poza tym wiemy jeszcze, że:

- Liczba 2 jest poprawną wartością.
- Liczba musi być większa od 1.
- Operacja x & 1 nie może zwracać 0.

Ponieważ chcemy uniknąć zwrotu do funkcji main wartości 0, przeanalizujemy logikę, która do tego prowadzi. Poznaną do tej pory logikę możemy przedstawić za pomocą następującego pseudokodu:

```
if ( x == 2 )
  return 1;

if (x <= 1 || (x & 1) == 0)
  return 0;
```

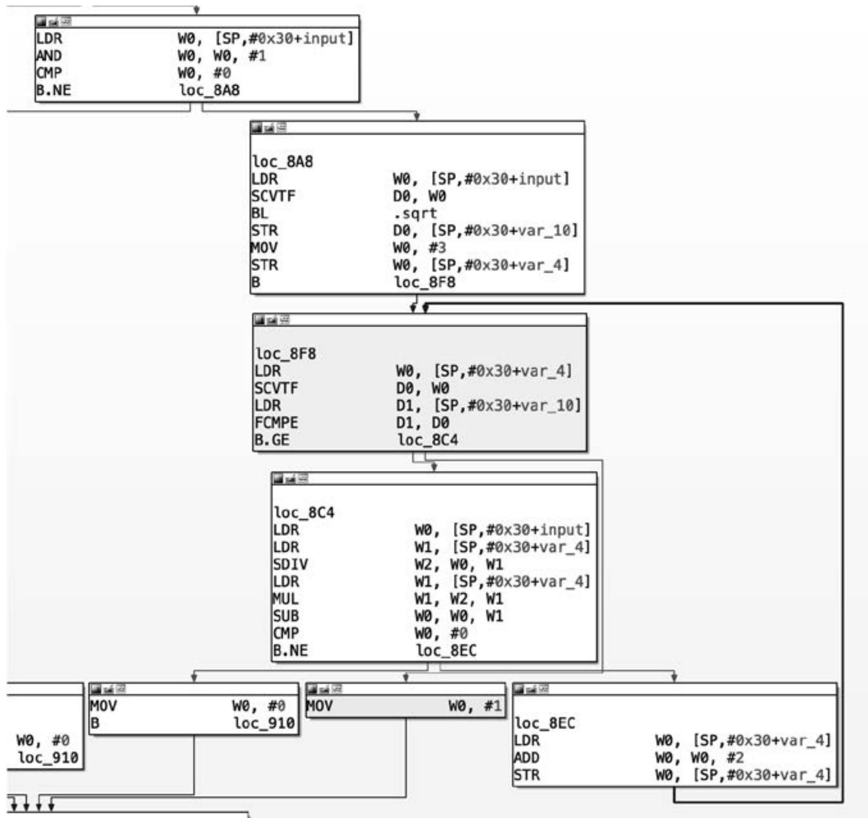
Ewentualnie możemy przedstawić ją tak:

```
if ( x == 2 )
  return 1;

if (x <= 1 || (x % 2) == 0)
  return 0;
```

Wiemy, że 2 jest jedną z poprawnych liczb. Jakie jeszcze warunki musi spełniać liczba, aby funkcja zwróciła wartość 1?

Kup ksi k Pole ksi k

Zaczniemy od końca funkcji `algoFunc` i prześledzimy, w jaki sposób dochodzimy do tego, że otrzymujemy wartość zwrotną 1. Na rysunku 10.44 widać, że aby dojść do bloku ustawiającego wartość zwrotną na 1, musimy wykonać rozgałęzienie do bloku instrukcji `loc_8A8`, który przechodzi do gałęzi `loc_8F8`. Tam gałąź `B.GE` musi zwrócić fałsz (strzałka w prawo) i przejść do rozkazu `MOV`, który ustawia wartość zwrotną na 1. W tym momencie nie wiemy, jak tam dojść, więc przeanalizujemy to krok po kroku.



Rysunek 10.44. Rozgałęzienie do bloku instrukcji loc_8A8

Pamiętasz operację `AND 1`, w której nasza liczba (14) okazała się niepoprawna? Teraz wybierzemy inną liczbę do obliczeń. W tym przypadku liczba 13 spełniłaby kryteria, które odkryliśmy do tej pory, ponieważ `13 & 1 = 1`. Trafiamy do bloku instrukcji `loc_8A8`, gdzie znajdujemy rozkaz, którego nie znamy: `SCVTF`.

W tej książce omówiłam większość najczęściej używanych rozkazów, ale nadal od czasu do czasu będziesz napotykać nowe. Wtedy dobrze jest mieć pod ręką podręcznik Arm. Znajdziemy w nim dwie wersje rozkazu `SCVTF`[8] (zobacz rysunek 10.45).

[8] *C3-242*, tabela C3-67, „Rozkazy konwersji liczb zmiennoprzecinkowych i całkowitych lub liczb stałoprzecinkowych”.

Kup ksi k Pole ksi k

SCVTF	Konwersja skalarnej wartości całkowitoliczbowej ze znakiem na wartość zmiennoprzecinkową przy użyciu bieżącego trybu zaokrąglania (forma skalarna)	*SCVTF (scalar, integer)* na str. C7-1931
	Konwersja liczby stałoprzecinkowej ze znakiem na liczbę zmiennoprzecinkową przy użyciu bieżącego trybu zaokrąglania (forma skalarna)	*SCVTF (scalar, fixed point)* na str. C7-1929

Rysunek 10.45. Rozkaz SCVTF

Jednym ze sposobów na rozgryzienie, który z tych dwóch wariantów rozkazu jest właściwy, jest przyjrzenie się składni. W naszym przypadku rozkaz ten używa `D0` jako rejestru docelowego i `W0` jako rejestru źródłowego, bez wartości bezpośredniej w składni. Oznacza to, że mamy do czynienia z wersją `SCVTF (scalar, integer)` i jej 32-bitowym wariantem o podwójnej precyzji. Oto jej opis[9]:

> *SCVTF (scalar, integer)*
>
> *Konwersja liczby całkowitej ze znakiem na liczbę zmiennoprzecinkową (skalar). Ten rozkaz konwertuje wartość całkowitoliczbową ze znakiem w rejestrze źródłowym ogólnego przeznaczenia na wartość zmiennoprzecinkową przy użyciu trybu zaokrąglania określonego przez* `FPCR` *oraz zapisuje wynik do rejestru docelowego* `SIMD&FP`.

Rejestrów zmiennoprzecinkowych jest 32 i są one ponumerowane od 0 do 31. Mogą być oznaczane jako `Q0`, `D0`, `S0` lub `H0`. W naszym przypadku `D0` reprezentuje 64-bitową wartość `double` i `long double` języka C. Choć wydaje się to dość skomplikowane, nie musimy zagłębiać się w szczegóły, aby zrozumieć, o co tu chodzi. Wiemy, że `SCVTF` to rozkaz konwersji zmiennoprzecinkowej, który konwertuje liczbę całkowitą ze znakiem z rejestru źródłowego (`W0`) na 64-bitową wartość podwójną. To ma sens, kiedy zauważymy, że następna instrukcja jest wywołaniem funkcji `sqrt`, która oblicza pierwiastek kwadratowy liczby wejściowej i zwraca wynik w postaci liczby zmiennoprzecinkowej. Nazwę tego bloku instrukcji możemy zmienić na `compute_sqrt`, a etykietę `var_10` na `sqrtX`, ponieważ wiemy, że w niej jest zapisywany wynik.

Wynikiem wywołania `sqrt(13)` jest wartość `3.60`, która zostaje zapisana na stosie. Jak widać na rysunku 10.46, rozkaz `MOV` ustawia rejestr `W0` na 3 (bez związku z naszym wynikiem) i zapisuje tę wartość na stosie. Kontynuujemy przez rozgałęzienie warunkowe do następnego bloku instrukcji, którym jest `loc_8F8`.

Dochodzimy do kolejnego rozkazu `SCVTF`, który konwertuje wartość `3` na format zmiennoprzecinkowy. Rozkaz `LDR` ładuje poprzedni wynik zwrócony przez funkcję `sqrt` do `D1`, po czym wykonuje rozkaz `FCMPE`[10], który porównuje liczby zmiennoprzecinkowe znajdujące się w rejestrach `D0` i `D1`. Jeśli wartość zwrócona przez funkcję `sqrt` jest nie mniejsza (`GE`) od 3, przechodzimy do następnego bloku instrukcji. Oto podsumowanie kroków:

- Zapisanie wyniku wywołania `sqrt(13)` na stosie.
- Zapisanie liczby całkowitej `3` na stosie.
- Konwersja liczby całkowitej `3` na format zmiennoprzecinkowy.
- Porównanie zmiennoprzecinkowych wersji wartości zwrotnej funkcji `sqrt` i liczby `3`.
- Jeśli wartość funkcji jest nie mniejsza od trójki, przejście do `loc_8C4`.

[9] *C7.2.236 SCVTF (scalar, integer).*

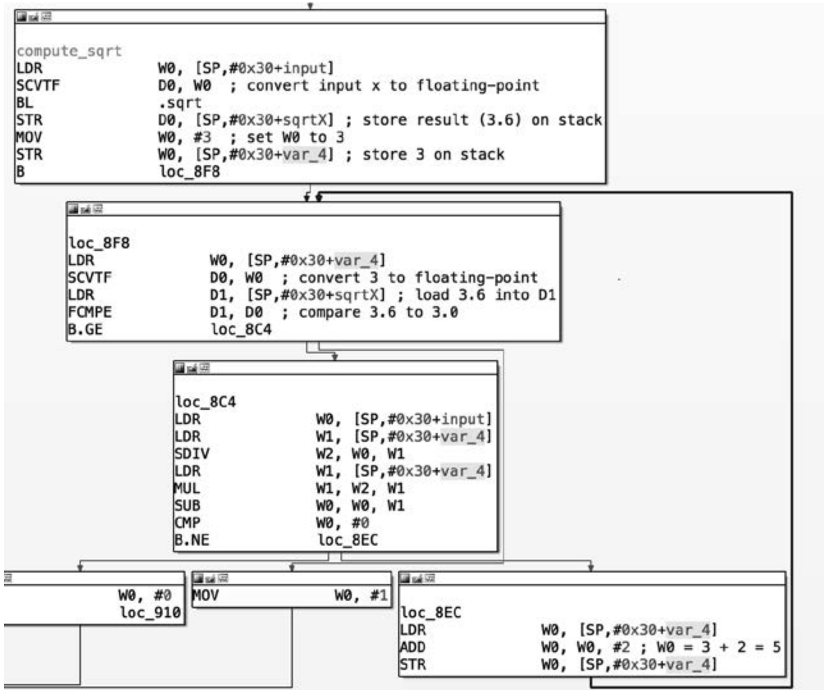
[10] *C7.2.67 FCMPE.*

Kup ksi k Pole ksi k

```
compute_sqrt
LDR             W0, [SP,#0x30+input]
SCVTF           D0, W0   ; convert input x to floating-point
BL              .sqrt
STR             D0, [SP,#0x30+sqrtX] ; store result (3.6) on stack
MOV             W0, #3
STR             W0, [SP,#0x30+var_4] ; store 3 on stack
B               loc_8F8

loc_8F8
LDR             W0, [SP,#0x30+var_4]
SCVTF           D0, W0   ; convert 3 to floating-point
LDR             D1, [SP,#0x30+sqrtX] ; load 3.6 into D1
FCMPE           D1, D0   ; compare 3.6 to 3.0
B.GE            loc_8C4
```

Rysunek 10.46. Blok instrukcji obliczający sqrt

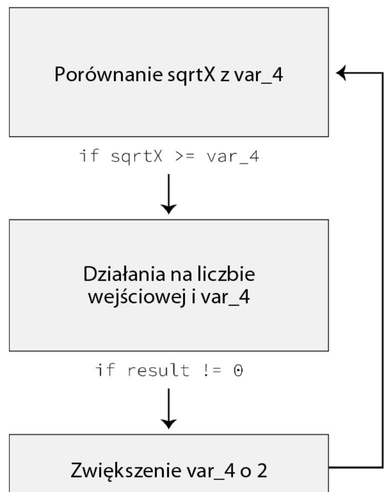
Spróbujmy rozgryźć cel porównywania wyniku z liczbą 3. Do czego jest używana ta wartość? Na schemacie bloków instrukcji z rysunku 10.47 widzimy, że wartość przechowywana w lokalizacji `var_4` jest używana jeszcze w paru innych przypadkach.



Rysunek 10.47. Inne przypadki użycia wartości var_4

Zaczniemy od bloku instrukcji `loc_8F8`. Na jego końcu znajduje się operacja porównywania, po której następuje rozgałęzienie uzależnione od tego, czy wartość `sqrtX` nie jest mniejsza (`GE`) od wartości `var_4`. Jeśli to prawda, wchodzimy do bloku instrukcji `loc_8C4`, który wykonuje pewne obliczenia na wartości wejściowej i wartości `var_4`. Jeśli wynik tego działania jest różny (`NE`)

od zera, przechodzimy do `loc_8EC` (blok na dole po prawej). Jedynym celem tego bloku jest zwiększenie wartości `var_4` o 2 i natychmiastowy powrót do bloku instrukcji `loc_8F8`. Bez zagłębiania się w szczegóły na temat tego, co się dzieje po drodze, otrzymujemy logikę pokazaną na rysunku 10.48.



Rysunek 10.48. Logika

Czy `var_4` może być licznikiem pętli `for`? To ma sens, ponieważ blok `compute_sqrt` ustawia `var_4` na stałą wartość (3), która jest następnie przetwarzana i zwiększana aż do spełnienia określonego warunku. Ten warunek można przedstawić następująco:

```
For (i = 3; sqrtX >= i; i += 2)
```

Aby zwiększyć czytelność, możemy zmienić nazwy `loc_8F8` na `for_loop_condition`, `loc_8EC` na `increment_i` oraz `var_4` na `i`. Pamiętaj, że w programie IDA Pro bloki instrukcji pętlowych są oznaczane niebieskimi grubymi strzałkami. W tym przypadku taka strzałka znajduje się po prawej stronie i prowadzi od bloku `increment_i` do bloku `for_loop_condition`.

Teraz przejdziemy do logiki bloku instrukcji `loc_8C4`. Zaczyna się on od ustawienia `W0` na wartość zmiennej `input` oraz `W1` na wartość `i` (zobacz rysunek 10.49).

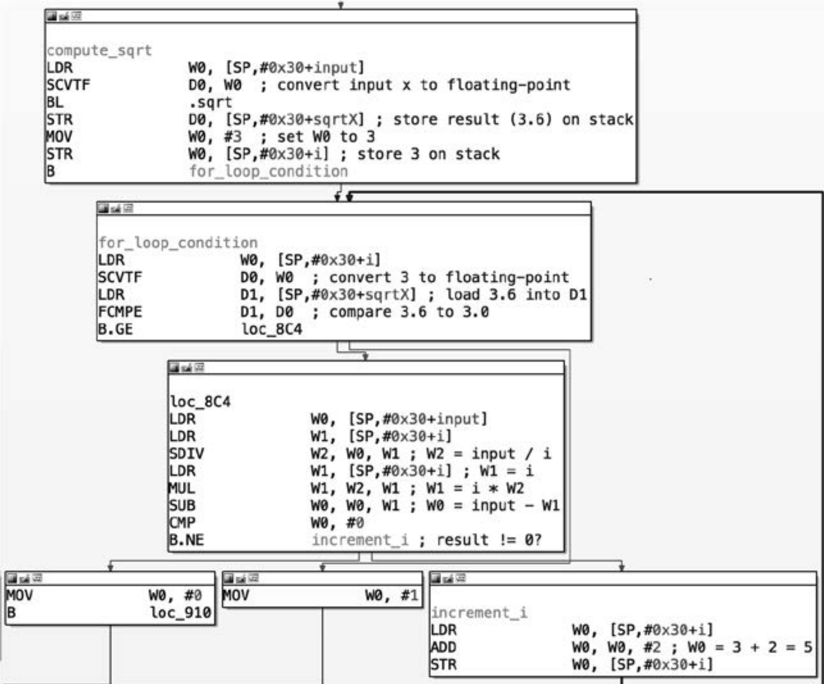
Rozkaz `SDIV` dzieli wartość `input` (`W0`) przez `i` (`W1`) i zapisuje wynik w rejestrze docelowym `W2`. Następnie rozkaz `MUL` mnoży wynik tego działania przez `i` oraz zapisuje iloczyn w rejestrze docelowym `W1`. Potem ten iloczyn zostaje odjęty od naszej wartości wejściowej i wynik zostaje zapisany w rejestrze `W0`. W naszym przykładzie te obliczenia wyglądają następująco:

```
13 / 3 = 4 ; X = input / i
4 * 3 = 12 ; Y = X * i
13 - 12 = 1 ; Z = input - Y
```

Blok instrukcji `loc_8C4` kończy się rozkazem `CMP` i rozgałęzieniem warunkowym. Sprawdza on, czy wynik (`W0`) różni się (`NE`) od `0`, a jeśli tak, przechodzi do bloku `increment_i`. Inaczej tę logikę można podsumować w następujący sposób:

```
if ( x == x / i * i )
return 0;
```

Kup książkę Poleć książkę



Rysunek 10.49. Blok instrukcji loc_8C4 z kontekstem

Pamiętaj, że ten blok instrukcji nie operuje na wartościach zmiennoprzecinkowych. Gdyby operował, wynik byłby inny.

Jeśli znane Ci są operacje dzielenia modulo, to szybko się zorientujesz, o co tu chodzi. Powyższa logika jest równoważna z poniższą:

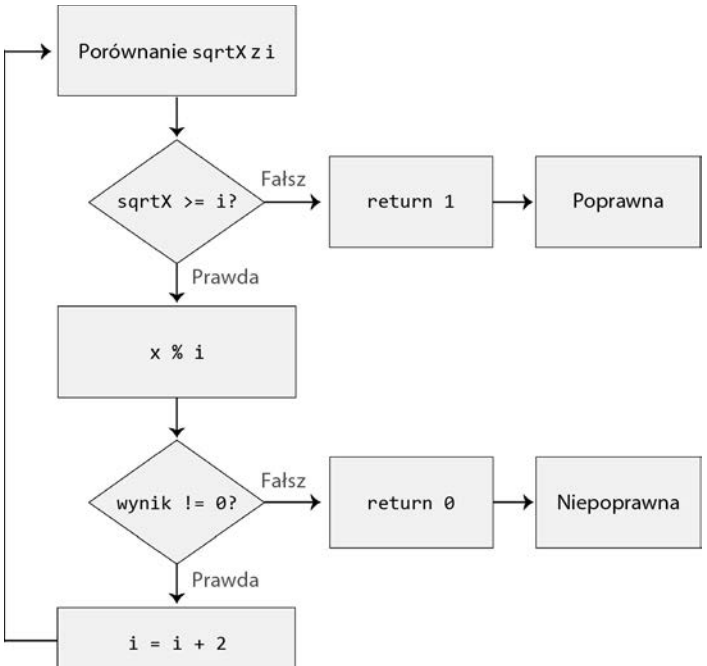
```
if ( x % i == 0 )
  return 0;
```

Jeśli wynik wynosi `0`, funkcja zwraca wartość `0`. Jeśli wynik jest inny niż `0`, funkcja zwiększa `i` o 2 i przechodzi do bloku warunkowego `for_loop_condition`. Ten blok porównuje wartość `sqrt(x)` (`3.6`) z nową wartością licznika (5), obie zmiennoprzecinkowe, po czym kontynuuje, jeśli wartość `sqrt(x)` nie jest mniejsza od nowej wartości licznika (5):

```
sqrtX = sqrt(x)
for (i = 3; sqrtX >= (double)i; i += 2){
    if (x % i == 0)
        return 0;
}
```

Liczba 3,6 nie jest większa od 5,0, więc przechodzimy do rozkazu `MOV`, który ustawia wartość zwrotną na `1`. Logikę tych bloków instrukcji możemy podsumować za pomocą schematu blokowego przedstawionego na rysunku 10.50.

Kup ksi k Pole ksi k



Rysunek 10.50. Logika

Zbierzmy wszystkie elementy w jedną całość i przedstawmy je w postaci pseudokodu:

```
algoFunc(int x)
{
    double sqrtX;
    int i;

    if ( x == 2 )
        return 1;
    if ( x <= 1 || (x & 1) == 0 )
        return 0;

    sqrtX = sqrt((double)x)

    for ( i = 3; sqrtX >= (double)i; i += 2 )
    {
        if ( x % i == 0 )
            return 0;
    }
    return 1;
}
```

Innymi słowy wartość zostaje uznana za poprawną, gdy jest większa od 1 oraz dzieli się bez reszty tylko przez 1 i przez siebie samą. Brzmi znajomo? Pewnie już się domyślasz, że jest to algorytm sprawdzający, czy dana liczba jest liczbą pierwszą.

Kup ksi k Pole ksi k

Sprawdźmy to, podając parę liczb pierwszych i jedną inną:

```
user@arm64:~$ ./algo1
Pick a number: 13
The answer is Yes!
user@arm64:~$ ./algo1
Pick a number: 17
The answer is Yes!
user@arm64:~$ ./algo1
Pick a number: 19
The answer is Yes!
user@arm64:~$ ./algo1
Pick a number: 20
20 does not meet the conditions.
```

Skorowidz

A

AArch32, 115
bity trybu, 126
licznik programu, 120
przełączanie między zestawami rozkazów, 117
rejestr, 120
bieżącego stanu programu, 122
połączenia, 121
stanu bloku IT, 125
stanu programu, 122
stanu wykonywania, 124
stanu zestawu rozkazów, 125
wywołań międzyproceduralnych, 122
stan wykonywania, 115
wskaźnik
ramki, 121
stosu, 121
zestaw rozkazów
A32, 116
T32, 116
AArch64, 106
licznik programu, 109
rejestr, 107
łączenia, 111
platformy, 111
zerowy, 110
rejestry
SIMD, 112
systemowe, 113
wywołań międzyproceduralnych, 112
zmiennoprzecinkowe, 112
stan wykonywania, 106
wskaźnik
ramki, 111
stosu, 109
zestaw rozkazów, 106
adres
bazowy wektorów, 103
zwrotny, 268
adresowanie, 187
literałowe, 202
postindeksowane, 200, 201
z offsetem, 188, 190
adresy
fizyczne, 86
wirtualne, 86
współpracy, 119
akumulacja, 163
wartości długich, 174
ze znakiem, 171
aliasy rozkazów, 130
analiza
algorytmu, 322
dynamiczna, 335
luk bezpieczeństwa, 357

analiza
przepływu sterowania, 309
statyczna, 297
narzędzia, 298
plików binarnych, 285
złośliwego oprogramowania, 383, 385
API sysctl, 388
architektura
Arm, 20, 97
Armv8-A, 99
procesora, 40
systemu operacyjnego, 76
Arm, 20, 97, 285
emulator QEMU, 287
płytki, 286
arm64
analiza złośliwego oprogramowania, 383
pliki binarne, 372
złośliwe pliki binarne, 377
Armv8-A, 99
poziomy wyjątków, 99
rejestr PSTATE, 114
rozszerzenie TrustZone, 100
stany wykonywania, 104
zmiany poziomu wyjątków, 102
artefakty maszyny wirtualnej, 391
ASCII, 23, 315
asemblacja, 26, 41
asembler, 20, 21, 24–27
skrośny, cross assembler, 31
ASLR, address space layout randomization, 64, 92
atrybuty wiązania, 60

B

bajt, 22
biblioteka libc, 80
binarne czyszczenie bitów, 151
binarny interfejs aplikacji, ABI, 270
Binary Ninja, 298
funkcja main, 299
funkcja Triage, 301
opcje wyświetlania, 300
bity, 22, 138
binarne czyszczenie, 151
maski wyjątku, 128
trybu, 126
blok sterowania wątku, TCB, 71
błąd
pamięci, 357
wyrównania do stosu, 186

D

debuger GNU Debugger, 336
debugowanie, 385, 388
błędu pamięci, 357
procesu, 365
zdalne, 356
złośliwego oprogramowania, 385, 388
dekompilacja, 33
dekompilator, 298
Ghidra, 34
IDA Pro, 34
dekrementacja
po, 218
przed, 218
dezasemblacja, 32, 319–324
funkcji write, 80
dezasembler, 298
Hopper, 385
dodawanie, 155
DRM, Digital Rights Management, 100
dyrektywa NEEDED, 62
dyrektywy asemblera, 27
dzielenie, 178
bez znaku, 178
współczynnika, 318
ze znakiem, 178

E

emulator QEMU, 287
etykiety, 27
exploit, 365

F

flagi warunkowe NZCV, 228
formy offsetu, 189
funkcja, 269
algoFunc, 325, 326
dlsym, 386
heap-analysis-helper, 347
main, 324
objc_msgSend, 394
scanf, 324

secret, 364
sysctl, 389
funkcje
epilog, 277, 280
liście, 277
niebędące liśćmi, 277
prolog, 277, 279

G

GDB
analizatory luk bezpieczeństwa, 347
badanie pamięci, 344
debugowanie
procesu, 365
zdalne, 356
instalacja GEF, 340
konfiguracja, 338
oglądanie obszarów pamięci, 346
polecenia, 337
GEF, 341, 342
rozszerzenie GEF, 340
Ghidra, 299
GOT, Global Offset Table, 65

I

IDA Pro, 298
blok instrukcji
loc_8A0, 327
loc_8A8, 328
loc_8C4, 332
debugowanie
błędu pamięci, 358
zdalne, 356
etykiety zmiennych lokalnych, 326
funkcja
decimal2Hexadecimal, 311, 312
main, 310
identyfikator procesu, PID, 77
implementacje stosu, 94
inicjalizacja, 66, 68
dynamiczna, 66
inkrementacja
po, 218
przed, 218
instrukcja
if, 316
if-else, 259, 314
inżynieria wsteczna, 21, 283, 370

J

jednostka MMU, 86
język asemblera, 27
języki
niskiego poziomu, 31, 37, 302
średniego poziomu, 302
wysokiego poziomu, 31, 37, 303

K

kod
binarny, 25
maszynowy, 21, 24
niezależny od pozycji, 64
szesnastkowy, 25
źródłowy, 36
kodowanie znaków, 23
kody warunkowe, 228, 231, 234
kompilacja, 37, 39
kompilator, 37
GCC, 34
kompilatory skrośne GCC, 41
konfiguracja GDB, 338
kontrola przepływu sterowania, 256
konwencja wywoływania, 270
konwersja na char, 315

L

leniwe wiązanie symboli, 65
licznik programu, 109, 120, 215
linkowanie, 41, 63
dynamiczne, 68
listingi asemblera, 41
loader, 62, 68
luka bezpieczeństwa, 357, 365

Ł

ładowanie
adresu do rejestru, 205
dynamiczne, 61
par, 223
półsłów, 210
słów, 208
stałych, 202
wielu wartości, 214, 215
zależności, 62

M

macOS, 370
  pliki binarne arm64, 372
  program powitalny, 375
maski
  bitowe, 149
  wyjątków, 126
maszyna wirtualna
  izolowana, 391
mikroarchitektura, 97
mnemonik, 25
mnożenie, 160, 163, 164
  -akumulowanie półsłów, 167
  -odejmowanie słów, 166
  podwójne, 170, 171, 176
  półsłów, 166, 167
  półsłów ze znakiem, 167
  słów przez półsłowa, 168
  w zestawach A32/T32, 162
  w zestawie A64, 160
  wartości długich, 172
  wektorów, 169
  z akumulacją długich półsłów, 175
  z dwukrotną akumulacją, 174
model dostępu TSL
  general-dynamic, 73
  initial-exec, 72
  local-dynamic, 75
  local-exec, 72
moduły mapowane, 89
modyfikatory wyszukiwania, 379
monitor bezpieczeństwa, 101

N

nagłówki
  programu ELF, 47
  sekcji pliku ELF, 54
narzędzia
  do analizy statycznej, 298
  wiersza poleceń, 298
narzędzie, *Patrz także* polecenie
  clang, 375
  csrutil, 392
  lldb, 385
  netstat, 295
  objdump, 322

O

obiekt, 83
  pliku, 84
obraz inicjalizacji TLS, 70
obrót, 131
  w prawo, 133
  w prawo z przeniesieniem, 134
odejmowanie, 155, 164
  odwrotne, 157
offset, 187
offsety rejestrowe, 196
opcje RELRO, 54
operacja
  binarna
    AND, 149
    OR, 151
    OR NOT, 152
    OR NOT wykluczającego, 154
    OR wykluczającego, 153
  dodawania, 155
  obrotu, 131
  odejmowania, 155
  odejmowania odwrotnego, 157
  pobierania, 148
  przesunięcia, 131
  wstawiania, 148
operacje
  arytmetyczne, 155
  dzielenia, 178
  logiczne, 149
  manipulacji bitami, 138
  mnożenia, 160, 163, 164, 167, 174
  przeniesienia, 179
  rozszerzenia, 143
  warunkowe
    z logicznym AND, 250
    z logicznym OR, 253
operator
  dereferencji *, 305
  referencji &, 305

P

pamięć
  anonimowa, 89
  lokalna wątków, 68
  mapowana, 89
  niezmapowana, 87
  współdzielona, 96

Kup ksi k Pole ksi k

pętla, 257
do-while, 259
for, 261, 320
while, 259, 319
pliki
binarne Arm
analiza statyczna, 285
binarne arm64, 372
złośliwe, 377
mapowane, 89
obiektowe, 41, 44
pliki ELF, 36, 42
ładowanie
dynamiczne, 61
zależności, 62
metasekcje, 56
nagłówek, 44
nagłówki sekcji, 54
obraz inicjalizacji TLS, 70
pola
informacyjne nagłówka, 45
lokalizacji tabel, 47
platformy docelowej, 46
pole punktu wejściowego, 46
sekcja
.bss, 57
.data, 57
.rodata, 58
.tbss, 58
.tdata, 58
.text, 57
dynamiczna, 61
inicjalizacji programu, 66
tabeli łańcuchów, 56
tabeli symboli, 57
terminacji programu, 66
struktura, 44
symbole, 58
typy relokacji TLS, 74
pobieranie pól bitowych, 147
podpisywanie kodu, 89
podprocedury, 267, 269, 311
pola bitowe
pobieranie, 147
wstawianie, 147
polecenia
GDB, 337
GEF, 341, 342
checksec, 349
memory watch, 346
polecenie
atop, 79
csrutil, 392
objdump, 33, 322
posix_spawn, 392
ps, 83
porównywanie, 157, 240
negatywne, 158, 242
warunkowe, 250
poziom
uprawnień, privilege level, 99
wyjątków, exception level, 99
procedura obsługi wyjątków, 127
proces kompilacji, 39
procesor Arm, 20
procesy, 77
profil, 98
program
DYNAMIC
nagłówek, 50
ELF
nagłówki, 47
Ghidra, 35
GNU_EH_FRAME
nagłówek, 51
GNU_RELRO
nagłówek, 53
GNU_STACK
nagłówek, 51
INTERP
nagłówek, 48
LOAD
nagłówki, 49
NOTE
nagłówek, 50
PHDR
nagłówek, 48
TLS
nagłówek, 50
przekazywanie większych wartości, 274
przeniesienie, 230
rejestru, 182
wartości bezpośredniej, 179, 181
wartości stałej, 180
z negacją, 184
przepełnienie
bez znaku, 230, 236
całkowite, 229
ze znakiem, 230, 236, 237
całkowitoliczbowe, 229

przepływ sterowania, 28, 256
przestrzeń adresowa, 92
przesunięcie, 131
arytmetyczne w prawo, 133
logiczne w lewo, 132
logiczne w prawo, 132
o wartość stałą, 135
o wartość zapisaną w rejestrze, 137
pola bitowego, 139
rejestrowe, 179
relokacyjne, relocation bias, 64
pula literałów, 203

Q

QEMU, 287
emulacja
oprogramowania układowego, 292
w trybie użytkownika, 288
pełna systemu, 291

R

Radare2, 299, 351
debugowanie, 351
debugowanie zdalne, 356
tryb wizualny, 353
widok interaktywny, 352
Raspberry Pi OS, 287
referencja, 304
rejestr, 25, 107, 120
adresu bazowego wektorów, 103
bazowy, 185
bieżącego stanu programu, 122
łączenia, 111, 121
platformy, 111
PSTATE, 114
stanu
bloku IT, 125
programu, 122
wykonywania, 124
zestawu rozkazów, 125
transferu, 186
transferu rozkazu, 185
wywołań międzyproceduralnych, 122
zerowy, 110
rejestry
AArch32, 120
AArch64, 107
argumentów, 274
bankowane, banked registers, 120
nieulotne, 271
ogólnego przeznaczenia, 271
stanu wykonywania, 124
systemowe, 113
ulotne, 271
zmiennoprzecinkowe, 112
relokacje, 49, 63, 73
dynamiczne, 63, 64
lokalne wątków, 63
statyczne, 63
rozgałęzienia
do podprocedur, 267
do rejestru, 257, 260
tabelowe, 262
warunkowe, 257, 314, 325
współpracy, 118
z łączem, 261, 268
z wartością bezpośrednią, 257
rozgałęzienie i zamiana, 264
rozkaz
ADD, 130, 146, 156
ADDS, 237
ADR, 29, 205
AND, 150
BLX, 264, 268
BR, 260
CBNZ, 262
CBZ, 262
CCMP, 250, 253–255
CMN, 158, 242
CMP, 157, 241, 249
CSEL, 249, 319
EXTR, 141
IT, 233
LDM, 215, 221, 272
LDP, 224, 226
LDPSW, 226
LDR, 29, 186, 203
LDRB, 194
LDRH, 194
MLA, 163, 164
MLS, 164
MOV, 180, 182, 204, 205
MOVK, 181, 182
MOVNE, 245
MOVT, 180
MOVZ, 181
MUL, 163

MVN, 184
OR, 152
OR NOT, 153
OR NOT wykluczający, 154
POP, 219
PUSH, 219
RBS, 157
ROR, 143
SCVTF, 329
SMLAD, 171
SMLAL, 174
SMLALD, 176
SMLALxx, 175
SMLSD, 171
SMLSLD, 178
SMMLA, 165
SMMLS, 166
SMMUL, 165
SMUAD, 170
SMUADX, 170
SMULL, 172
SMULWB, 168
SMULWT, 168
SMUSD, 170
STM, 215–217, 221
STP, 224, 225
STR, 186
SUB, 156
SVC, 387
TEQ, 154, 246
TST, 150, 243–245
UMAAL, 174
UMLAL, 174
UMULL, 172
rozkazy
binarnego czyszczenia bitów, 151
dostępu do pamięci, 185
kodowanie, 266
ładowania, 208
i zapisu, 213
i zapisu bajta, 212
i zapisu półsłów, 211
słowa, 210
mnożenia, 239
i akumulacji, 174
i akumulacji słowa przez półsłowo, 169
i odejmowania, 164
słów, 168
o rozszerzonym rejestrze, 145
pobierania pól bitowych, 147
porównywania, 157, 240, 262
warunkowego, 250
przeniesienia rejestrów, 183
przesunięcia
i obrotu, 137
logicznego, 238
pola bitowego, 139
przetwarzania danych, 129
rozgałęzienia, 256, 262
i zamiany, 264
warunkowego, 258
rozszerzenia A64, 143
rozszerzeń zestawów A32/T32, 146
testowania, 240
bitu i rozgałęzienia, 262
ustawiające flagi, 228, 235
warunkowe, 228, 232
wyboru warunkowego, 247, 248
wywołujące podprocedury, 268
z offsetem skalowanym, 194
z przyrostkiem S, 236
zapisu, 208
rozszerzanie
niejawne, 145
zerami i znakiem, 145
znakiem lub zerami, 143

S

sekcja dynamiczna, 61
stała wartość bezpośrednia, 191
stan wyjątkowy, 127
standard AAPCS, 270, 272
standardy ABI, 270
statyczna inicjalizacja, 66
stos, stack, 94, 109, 121, 311, 320
strony pamięci, 87
sygnatura funkcji, 269
symbole, 58
globalne, 60
lokalne, 60
mapowania, 61, 70
słabe, 60
typy, 59
wersjonowanie, 60
system operacyjny macOS, 370
systemy operacyjne
architektura, 76

Ś

- środowiska Arm, 285
- środowisko wykonawcze
  - bogate, REE, 101
  - zaufane, TEE, 101
- świat
  - normalny, normal world, 100
  - zabezpieczony, secure world, 100

T

- tabela
  - ASCII, 315
  - dynamiczna, 62
  - globalna przesunięć, GOT, 65
  - łańcuchów, 56
  - łączenia procedur, PLT, 65, 112
  - relokacji, 63
  - symboli, 57, 58
  - wektorów, vector table, 103
- tablica prawdy
  - operacji AND, 150
  - operacji NOT OR, 152
  - operacji OR, 151
- terminacja, 66, 68
- testowanie, 240
  - bitów, 243
- tryb
  - adresowania, 187
    - literałowego, 188, 202
    - offsetowego, 198
    - pojedynczego rejestru, 189
    - postindeksowany, 188, 200, 201
    - preindeksowany, 188, 198
  - jądra, 76
  - użytkownika, 76
  - użytkownika QEMU, 288
  - wyjątków, 127
- typy
  - danych, 273
  - relokacji TLS, 74
  - symboli, 59

U

- uchwyty, 83
- uprawnienia, 88
- użytkownik, 82

W

- wartości zwrotne, 272
- wątki, 68, 85
- wejście standardowe, stdin, 85
- wektor wyjątku, exception vector, 103
- wiersz poleceń
  - debugowanie, 336
  - narzędzia Radare2, 351
- wskaźnik, 304
  - ramki, 111
  - stosu, stack pointer, 94, 109
- współpraca, interworking, 105, 115
- wstawianie pól bitowych, 147, 148
- wybór warunkowy, 247
- wyjątek, 126
  - SVC, 79
  - wyrównania stosu, 110
  - wywołania nadzorcy, 81
- wyjątki synchroniczne, 102, 127
- wyjście standardowe błędów, stderr, 85
- wywołania systemowe, syscall, 78
- wywoływanie procedur, 270

Z

- zabezpieczenia pamięci, 88
- zapis
  - par, 223
  - półsłów, 210
  - słów, 208
  - wielu wartości, 214, 215
- zarządzanie pamięcią procesu, 86
- zaufany sterownik, TD, 101
- złośliwe oprogramowanie, 377, 381
  - analiza, 383–385
  - debugowanie, 385, 388
  - utrudnienia analizy, 384
  - zmienne lokalne wątków, 69

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

POZNAJ MOC INŻYNIERII WSTECZNEJ!

Procesory ARM są coraz popularniejsze w urządzeniach mobilnych, laptopach i serwerach. Rosnąca popularność czyni je interesującymi dla badaczy bezpieczeństwa. Specjaliści z tej branży często używają technik inżynierii wstecznej podczas badania plików binarnych. W tym celu konieczne jest zapoznanie się z poleceniami asemblera ARM.

Książkę szczególnie docenią analitycy bezpieczeństwa, którzy chcą poznać zestaw poleceń ARM i zdobyć wiedzę umożliwiającą im efektywne korzystanie z technik inżynierii wstecznej. Poza zestawem potrzebnych poleceń znalazło się w niej mnóstwo przydatnych informacji. Znajdziesz tu przegląd podstawowych zagadnień związanych z systemem operacyjnym, wyjaśnienie, czym są polecenia asemblera i na czym polega proces kompilacji pliku, a także opis mechanizmów działania poleceń służących do przetwarzania danych, dostępu do pamięci i kontroli przepływu sterowania. W kolejnych rozdziałach zapoznasz się z przeglądem technik inżynierii wstecznej, takich jak analiza statyczna i dynamiczna, jak również z kompleksowym omówieniem złośliwego oprogramowania, które może ją utrudniać.

W książce:

- wprowadzenie do architektury ARM
- zestawy poleceń: A64, A32 i T32, a także format plików ELF
- przegląd wzorców przepływu sterowania w procesorach ARM
- narzędzia inżynierii wstecznej
- proces dezasemblowania i debugowania plików binarnych ARM w systemie Linux
- typowe narzędzia do dezasemblowania i debugowania plików binarnych ARM

MARIA AZERIA MARKSTEDTER zajmuje się inżynierią wsteczną i podatnościami architektury ARM. Wcześniej prowadziła testy penetracyjne i badała zagrożenia cyberprzestrzeni. W 2020 roku magazyn „Forbes" przyznał jej tytuł Człowieka Roku w branży cyberbezpieczeństwa. Należy do komisji europejskiej edycji konferencji Black Hat®.

ISBN 978-83-289-0675-4