

# Architektury systemów komputerowych

## Lista zadań nr 9

Na zajęcia 6 i 7 maja 2024

Zadania z tej listy należy rozwiązywać na komputerze z systemem operacyjnym *Linux* dla platformy *x86-64*. Prowadzący zakłada, że zainstalowana dystrybucja będzie bazowała na *Debianie 11*. Do poniższej listy załączono na stronie przedmiotu pliki źródłowe wraz z plikiem *Makefile*.

**UWAGA!** W trakcie prezentacji rozwiązań należy zdefiniować i wyjaśnić pojęcia, które zostały oznaczone **wytluszczoną** czcionką.

**Zadanie 1.** Poniżej zamieszczono uproszczony wynik kompilacji pliku «data.c» do asemblera. Z jakimi **opcjami**<sup>1</sup> **sterownik kompilatora** (tj. polecenie *gcc*) wywołał kompilator języka C (tj. polecenie «cc1»), aby otrzymać plik «data.s»? Na podstawie dokumentu *GNU as: Assembler Directives*<sup>2</sup> wyjaśnij znaczenie **dyrektyw asemblera** użytych w poniższym kodzie. Które z dyrektyw przełączają bieżącą sekcję, dopisują zawartość do odpowiednich sekcji, modyfikują informacje przechowywane w nagłówku sekcji lub tablicę symboli?

```
1  .globl foo           14  .globl bar           29  .globl some
2  .data               15  .bss                30  .data
3  .align 2            16  .align 8            31  .align 32
4  .type foo, @object  17  .type bar, @object      32  .type some, @object
5  .size foo, 2        18  .size bar, 8        33  .size some, 38
6 foo:                19 bar:                34 some:
7  .value 314          20  .zero 8            35  .quad weird
8                      21                      36  .long -3
9  .section .rodata    22  .globl array        37  .zero 4
10 .type abc, @object  23  .bss                38  .quad abc
11 .size abc, 4        24  .align 32            39  .quad foo
12 abc:                25  .type array, @object    40  .string "efghi"
13 .string "abc"       26  .size array, 800
                      27 array:
                      28  .zero 800
```

**Zadanie 2.** Poniżej zamieszczono uproszczony wynik kompilacji plików «start.c», «odd.c» i «even.c» do asemblera. W wygenerowanych plikach wskaż miejsca występowania definicji symboli i referencji do symboli. Czemu asembler nie może wygenerować ostatecznego ciągu bajtów reprezentujących instrukcje «call» i «jmp»? Zweryfikuj to wyświetlając zdeasembrowany kod przy pomocy «objdump -d». Jakie informacje asembler zostawia w plikach relokowalnych, żeby konsolidator mógł uzupełnić te instrukcje? Przypomnij uczestnikom zajęć jakie są główne zadania pełnione przez konsolidator – posłuż się w tym celu **mapą konsolidacji** z pliku «start.map». Pokaż, że konsolidator uzupełnił wymienione instrukcje w pliku wykonywalnym «start».

```
1  .text               1  .text               1  .text
2  .globl _start       2  .globl is_odd        2  .globl is_even
3  .type _start, @function 3  .type is_odd, @function 3  .type is_even, @function
4  _start:             4  is_odd:             4  is_even:
5  pushq %rax          5  testq %rdi, %rdi      5  testq %rdi, %rdi
6  movl $42, %edi      6  je .L2              6  je .L2
7  call is_even        7  decq %rdi           7  decq %rdi
8  movl %eax, %edi     8  jmp is_even         8  jmp is_odd
9  movl $60, %eax      9  .L2:               9  .L2:
10 syscall            10 xorl %eax, %eax      10 movl $1, %eax
11 popq %rdx           11 ret                11 ret
12 ret                 12 .size is_odd, .-is_odd 12 .size is_even, .-is_even
13 .size _start, .-_start
```

<sup>1</sup><https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html>

<sup>2</sup><https://sourceware.org/binutils/docs/as/Pseudo-Ops.html>

**Zadanie 3.** Na kodzie z zadania 1 wykonaj proces tworzenia zawartości poszczególnych sekcji, który normalnie przeprowadzany jest przez *assembler*. Załóżmy, że zawartość sekcji `«.rodata»` zostanie zapisana w sekcji `«.data»`. Zauważ, że tworzenie zawartości sekcji `«.bss»` jest trywialne! Wyjaśnij co się dzieje w momencie przetwarzania każdego wiersza. Zawartość utworzonej sekcji `«.data»` należy zaprezentować podobnie jak polecenie `«objdump -s»`. Należy również utworzyć ręcznie tabelę **symboli** oraz **rekordów relokacji** i podać odpowiednio w postaci w jakiej drukuje je polecenie `«nm»` i `«objdump -r»`.

**Zadanie 4.** Przeprowadź na swoim komputerze atak na program `«ropex»` wykorzystując podatność **przepełnienia bufora** w procedurze `«echo»`. Posłuż się techniką **ROP** (ang. *return oriented programming*). Wyznacz adresy **gadżetów**, tj. procedury `«gadget»` oraz dowolnej instrukcji `«syscall»` w pliku `«ropex»`. Wpisz je, w porządku *little-endian*, do pliku `«ropex.in.txt»` na pozycji `0x38` i `0x40`, po czym przetłumacz go do postaci binarnej. Następnie uruchom polecenie `«ropex ropex.in»`, aby zobaczyć rezultat wykonania programu `«nyancat»`<sup>3</sup>. Przy pomocy `gdb` zaprezentuj zawartość stosu przed i po wykonaniu procedury `«gets»`. Pokaż, że procesor wykonując instrukcję `«ret»` skacze pod przygotowane przez Ciebie adresy.

**Wskazówka:** Wykaz poleceń i odnośnik do samouczka `gdb` podano na stronie przedmiotu w SKOS.

**Zadanie 5.** Zmodyfikuj opcje kompilacji programu `«ropex»` w pliku `«Makefile»`. Najpierw zleć kompilatorowi dodanie **kanarków** (ang. *canary*) włączając opcję `«-fstack-protector»`. Pokaż, że program wykrywa **uszkodzenie stosu** (ang. *stack smashing*). Posługując się debuggerem `gdb` zaprezentuj, że przy każdym uruchomieniu programu wartość kanarka jest inna. Następnie usuń opcję wymuszającą *statyczną konsolidację* `«-static»` i dodaj opcję `«-fpie»`, aby umożliwić **randomizację rozkładu przestrzeni adresowej** (ang. *Address Space Layout Randomization*). Pokaż, że adres *gadżetu* o nazwie `«gadget»` jest inny przy każdym uruchomieniu programu. Następnie dodaj opcję kompilacji `«-z noexecstack»`, która zapisuje w pliku ELF informacje o tym, że procesor nie powinien próbować wykonywać zawartości stosu. Przy pomocy programu `«pmap»` zweryfikuj, że istotnie stos uruchomionego programu `«ropex»` nie jest wykonywalny.

Czemu każde z zastosowanych wyżej zabezpieczeń utrudnia zadanie atakującemu?

**Uwaga!** Debugger `gdb` może wyłączyć ASLR przeprowadzaną przez konsolidator dynamiczny, aby ułatwić sobie pracę.

**Zadanie 6.** Jądro systemu Linux nie potrafi załadować do pamięci pliku wykonywalnego skonsolidowanego dynamicznie – musi o to poprosić **interpreter programu** [2, 5]. Rozważmy plik wykonywalny `«/bin/sleep»`. Na podstawie zawartości jego sekcji `«.interp»` podaj ścieżkę do **konsolidatora dynamicznego** [2, 5]. Przy pomocy polecenia `«nm»` wyświetl wszystkie *symbole dynamiczne* – `ld.so(8)` będzie musiał znaleźć ich definicje w bibliotekach dynamicznych. Przy pomocy polecenia `«readelf -d»` wyświetl sekcję `«.dynamic»` [2, 5-9] i wskaż biblioteki, w których będą wyszukiwane definicje symboli. Na podstawie podręcznika `ldconfig(8)` znajdź plik konfiguracyjny przechowujący ścieżki, gdzie konsolidator będzie szukał bibliotek. Wskaż skąd zostanie załadowana biblioteka `«libc.so.6»`. Przy pomocy polecenia `ldd(1)` wyświetl pod jakie adresy konsolidator załadowałby biblioteki, gdyby miał załadować program do pamięci. Czemu za każdym razem adresy bibliotek są inne?

**Zadanie 7.** Na podstawie [2, 5-9] oraz `ld.elf_so(1)` opisz zadania pełnione przez *konsolidator dynamiczny*. Wykonaj polecenie `LD_DEBUG=all /bin/sleep 1»`. Wskaż w wydruku proces wyszukiwania i ładowania bibliotek, **wiązania symboli** (ang. *symbol resolution*) w trakcie ładowania programu i po jego uruchomieniu.

**Zadanie 8.** Na podstawie [1, §7.12] opisz proces **leniwego wiązania** (ang. *lazy binding*) symboli. Czym różni się **kod relokalny** (ang. *Position Independent Code*) skompilowany z opcją `«-fpic»` od kodu nierelokalnego? Jakie dane przechowują sekcje **procedure linkage table** `«.plt»` i **global offset table** `«.got»`? W jaki sposób korzysta z nich *konsolidator dynamiczny*? Czemu sekcja `«.got»` jest modyfikowalna, a sekcje kodu i `«.plt»` są tylko do odczytu? Jakie są przewagi *leniwego wiązania* nad *gorliwym wiązaniem*?

**Zadanie 9.** Zapoznaj się z obrazkiem [1, 7.19], a następnie zaprezentuj *leniwe wiązanie* na podstawie programu `«lazy»`. Załaduj program do debuggера `gdb` i ustaw **punkty wstrzymania** (ang. *breakpoint*) na wiersz 4 i 5. Po **uruchomieniu** program powinien zatrzymać się we wskazanych miejscach. Zauważ, że za pierwszym wywołaniem procedury `«puts»` w `«.got»` jest zapisany inny adres niż za drugim wywołaniem. Pokaż to wykonując program **krokowo**. Gdzie procesor skacze przy pierwszym wywołaniu `«puts»`?

<sup>3</sup>Do pliku `«ropex.in.txt»` można wpisać inny program pod warunkiem, że jego ścieżka będzie nie dłuższa.

## Literatura

- [1] „*Computer Systems: A Programmer's Perspective*”  
Randal E. Bryant, David R. O'Hallaron; Pearson; 3rd edition, 2016
- [2] „*System V Application Binary Interface*”  
<http://www.sco.com/developers/gabi/latest/contents.html>
- [3] „*System V Application Binary Interface: AMD64 Architecture Processor Supplement*”  
<https://raw.githubusercontent.com/hjl-tools/x86-psABI/x86-64-psABI-1.0.pdf>