Architektury systemów komputerowych

Lista zadań nr 9

Na zajęcia 6 maja 2021

Zadania należy prezentować na komputerze z systemem operacyjnym Linux dla platformy x86-64. Prowadzący zakłada, że zainstalowana dystrybucja będzie bazowała na Debian-ie (np. Ubuntu lub Mint). Przed przystąpienie do rozwiązywania zadań należy zainstalować GDB dashboard[1] i upewnić się, że działa.

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 **wytłuszczoną** czcionką.

Zadanie 1. W trakcie tłumaczenia poniższego kodu na asembler kompilator umieścił tablicę skoków dla instrukcji wyboru `switch` w sekcji «`.rodata`». W sekcji «`.text`» i «`.rodata`» wskaż miejsca występowania referencji do symboli, a następnie podaj zawartość tablicy rekordów relokacji «`.rela.text`» i «`.rela.rodata`».

```
1  int relo3(int val) {
2    switch (val) {
3      case 100:
4        return val;
5      case 101:
6        return val + 1;
7      case 103:
8      case 104:
9        return val + 3;
10     case 105:
11       return val + 5;
12     default:
13       return val + 6;
14   }
15 }
```

```
0000000000000000 <relo3>:
 0: 8d 47 9c              lea    -0x64(%rdi),%eax
 3: 83 f8 05              cmp    $0x5,%eax
 6: 77 15                 ja     1d <relo3+0x1d>
 8: 89 c0                 mov    %eax,%eax
 a: ff 24 c5 00 00 00 00  jmpq   *0x0(,%rax,8)
11: 8d 47 01              lea    0x1(%rdi),%eax
14: c3                    retq
15: 8d 47 03              lea    0x3(%rdi),%eax
18: c3                    retq
19: 8d 47 05              lea    0x5(%rdi),%eax
1c: c3                    retq
1d: 8d 47 06              lea    0x6(%rdi),%eax
20: c3                    retq
21: 89 f8                 mov    %edi,%eax
23: c3                    retq
```

W wyniku konsolidacji pliku wykonywalnego zawierającego procedurę «`relo3`», została ona umieszczona pod adresem `0x1000`, a tablica skoków pod `0x2000`. Oblicz wartości, które należy wstawić w miejsca referencji, do których odnoszą się rekordy relokacji.

Zadanie 2. Język C++ pozwala na przeciążanie funkcji (ang. *function overloading*), tj. definiowanie wielu funkcji o tej samej nazwie, ale różnej sygnaturze. Konsolidator nie przypisuje typów języka programowania poszczególnym symbolom. Powstaje zatem problem unikatowej reprezentacji nazw używanych przez język C++. Na czym polega **dekorowanie nazw** (ang. *name mangling*)? Które elementy składni podlegają dekorowaniu? Przy pomocy narzędzia `c++filt` przekształć poniższe nazwy na sygnatury funkcji języka C++ i omów znaczenie poszczególnych fragmentów symbolu. Czy funkcja dekorująca nazwy jest różnowartościowa?

`_Z4funcPKcRi` `_ZN3Bar3bazEPc` `_ZN3BarC1ERKS_` `_ZN3foo6strlenER6string`

Wskazówka: Szczegółowe informacje na temat kodowania nazw można znaleźć w C++ ABI: External Names[2].

Zadanie 3. 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 procedury «`gadget`» oraz dowolnej instrukcji «`syscall`» w pliku «`ropex`». Wpisz je, w porządku little-endian, do pliku «`ropex.in.txt`» na pozycji $38 i $40, po czym przetłumacz go do postaci binarnej. Następnie uruchom polecenie «`ropex ropex.in`», aby zobaczyć rezultat wykonania programu «`nyancat`»[3] w terminalu. Przy pomocy debuggera gdb zaprezentuj zawartość stosu przed i po wykonaniu procedury «`gets`». Pokaż jak procesor wykonując instrukcje «`ret`» skacze pod przygotowane przez Ciebie adresy.

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

[1] `https://github.com/cyrus-and/gdb-dashboard`

[2] `https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling`

[3] Do pliku «`ropex.in.txt`» można wpisać inny program pod warunkiem, że jego ścieżka będzie nie dłuższa.

Zadanie 4. 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, w związku z czym atakującemu będzie ciężką ją zgadnąć. Następnie usuń opcję wymuszającą statyczną konsolidację «-static» i dodaj opcję «-fpie», aby odbezpieczyć **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 odbiera możliwości wykonywania zawartości stosu. Przy pomocy programu «pmap» zweryfikuj, że istotnie stos uruchomionego programu «ropex» jest niewykonywalny.

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

Zadanie 5. Podglądając wyjście z kompilatora języka C pokaż jak korzystając z **dyrektyw asemblera** opisanych w GNU as: Assembler Directives[4]:

- zdefiniować globalną funkcję foobar,
- zdefiniować lokalną strukturę podaną niżej:

```
static const struct {
  char a[3]; int b; long c; float pi;
} baz = { "abc", 42, -3, 1.4142 };
```

- zarezerwować miejsce dla tablicy long array[100].

Wyjaśnij znaczenie poszczególnych dyrektyw. Pamiętaj, że dla każdego zdefiniowanego symbolu należy uzupełnić odpowiednio tabelę «.symtab» o typ symbolu i rozmiar danych, do których odnosi się symbol.

Zadanie 6. Jądro systemu Linux nie potrafi samodzielnie załadować do pamięci pliku wykonywalnego skonsolidowanego dynamicznie. Musi o to poprosić **konsolidator dynamiczny** opisany w podręczniku systemowym ld.so(8). Na podstawie ld.elf_so(1) opisz proces ładowania pliku pliku wykonywalnego «/bin/sleep» do pamięci. Wyświetl jego sekcję «.dynamic» i wskaż pliki konfiguracyjne na podstawie których konsolidator odnajdzie na dysku bibliotekę «libc.so.6». 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 7 (2). Na podstawie [1, §7.12] opisz proces **leniwego wiązania** (ang. *lazy binding*) symboli i odpowiedz na następujące pytania:

- Czym charakteryzuje się **kod relokowalny** (ang. *Position Independent Code*)?
- Do czego służą sekcje «.plt» i «.got» – jakie dane są tam przechowywane?
- Czemu sekcja «.got» jest modyfikowalna, a sekcje kodu i «.plt» są tylko do odczytu?
- Co znajduje się w sekcji «.dynamic»?

Zaprezentuj leniwe wiązanie na podstawie programu «lazy». Uruchom go pod kontrolą debuggera gdb, ustaw punkty wstrzymań (ang. *breakpoint*) na linię 4 i 5. Po czym wykonując krokowo program (stepi) pokaż, że gdy procesor skacze do adresu procedury «puts» zapisanego w «.got» – za pierwszym wywołaniem jest tam przechowywany inny adres niż za drugim.

Wskazówka: Wykorzystaj rysunek 7.19.

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 AMD64 Architecture Processor Supplement"
H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, Mark Mitchell;
https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf

[4] https://sourceware.org/binutils/docs-2.26/as/Pseudo-Ops.html