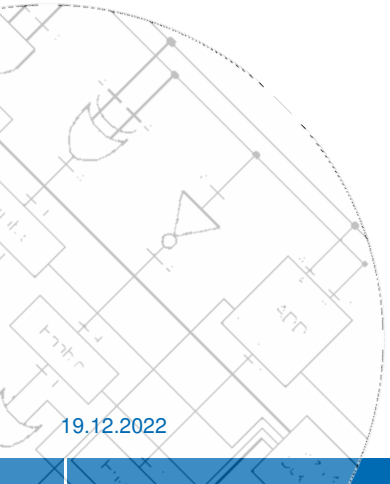
Rechnerarchitektur

Vorlesung 16: Funktionen, Von C zu Assembler, Buffer Overflow Exploits, Kapitelabschluss

Prof. Dr. Martin Mauve



19.12.2022

Haben Sie noch Fragen zur letzten Vorlesung?

Thema: Sprünge, Schleifen und der Stack

Funktionen

- Für die Unterstützung von Funktionen müssen drei wichtige Probleme gelöst werden:
  - Rücksprungadresse merken
  - Parameterübergabe regeln
  - Übergabe des Rückgabewertes regeln
- Der Instruktionssatz stellt Hilfsmittel zur Lösung dieser Probleme zur Verfügung.
- Die Regelung, wie diese Hilfsmittel angewendet werden, ist abhängig von Compiler und Programmiersprache.
- Man nennt diese Regelung „Aufrufkonvention“ oder „Calling Convention“.
- Wir betrachten im Folgenden die Calling Convention von C.
- Wenn man Assembler mit anderen Hochsprachen kombinieren will, muss man deren Calling Convention verwenden.

Rücksprungadresse: Erste Idee

- Im Prinzip könnte man einen Funktionsaufruf durch eine jmp-Instruktion realisieren.
- Problem: Merken, wohin nach Ende der Funktion zurückgekehrt werden soll.
- Idee:
  - Adresse der ersten Instruktion hinter dem Sprungbefehl in die Funktion in ein Register packen (via Label),
  - dann in die Funktion springen und
  - am Ende der Funktion an die gespeicherte Adresse springen.
- Umständlich: Label für jeden Rücksprungpunkt notwendig
- unintuitiv und fehleranfällig
- Größtes Problem: Was machen, wenn die Funktion wieder eine andere Funktion aufruft? Das Register ist schon belegt . . .

Rücksprungadresse: Nachgebessert

- Statt die Rücksprungadresse in ein Register zu schreiben, könnten wir sie auf den Stack packen.
- Also:
  - Adresse der ersten Instruktion hinter dem Sprung in die Funktion auf den Stack legen,
  - dann in die Funktion springen und
  - am Ende der Funktion die Rücksprungadresse vom Stack holen und dort hinspringen.
- Funktioniert auch bei geschachtelten Funktionsaufrufen.
- Ist aber immer noch ziemlich umständlich.
- Deshalb: „Komfortfunktionen“, die das für uns erledigen.

Rücksprungadresse: `call` und `ret`

- In Assembler wird eine Funktion mit `call` aufgerufen:
  - Beispiel: `call eingabe`
  - ESP wird um 4 verringert, die Rücksprungadresse (Adresse des Befehls nach `call`) wird an diese Stelle gelegt (wie bei `push`)
  - dann wird zur Instruktion mit dem Label eingabe gesprungen
  - benötigt kein Label mehr auf den Rücksprungpunkt
- In Assembler kehrt man mit `ret` aus einer Funktion zurück:
  - Beispiel: `ret`
  - Das Register EIP wird vom Stack geladen, ESP wird um 4 erhöht
    - EIP ist der Extended Instruction Pointer, er zeigt auf den als nächstes auszuführenden Befehl.
  - Dazu muss natürlich der ESP auf die richtige Position im Stack zeigen:
    - aufpassen, wenn die Funktion den Stack selbst verwendet!

Beispiel: Funktionsaufrufe I

```
      mov eax, 0x10      ; Die größere der beiden Zahlen
      mov ebx, 0x20      ; soll in ecx gespeichert werden
      dump_regs 1        ; Registerwerte ausgeben
      dump_stack2 1,0,2 ; Stack ausgeben
      call max
      dump_regs 2        ; Registerwerte ausgeben
      ...
max:  dump_stack2 2,0,2 ; Stack ausgeben
      cmp eax, ebx       ; Der Vergleich
      jl bgr
      mov ecx, eax
      jmp ende
bgr:  mov ecx, ebx
ende: ret
```

(*dump_stack2* ist nicht in *asm_io.asm* enthalten.)

Beispiel: Funktionsaufrufe II

Ausgabe:

```
Register Dump # 1
EAX = 00000010 EBX = 00000020 ECX = 401579A8 EDX = 40158E90
ESI = 40014580 EDI = BFFFF544 EBP = BFFFF4E8 ESP = BFFFF4C8
EIP = 0804843F FLAGS = 200286       SF       PF
Stack Dump # 1
EBP = BFFFF4E8 ESP = BFFFF4C8
  +8  BFFFF4D0  BFFFF4E8
  +4  BFFFF4CC  40014580
  +0  BFFFF4C8  BFFFF544
Stack Dump # 2
EBP = BFFFF4E8 ESP = BFFFF4C4
  +8  BFFFF4CC  40014580
  +4  BFFFF4C8  BFFFF544
  +0  BFFFF4C4  08048462
Register Dump # 2
EAX = 00000010 EBX = 00000020 ECX = 00000020 EDX = 40158E90
ESI = 40014580 EDI = BFFFF544 EBP = BFFFF4E8 ESP = BFFFF4C8
EIP = 08048462 FLAGS = 200287       SF       PF CF
```

Parameterübergabe

- Bei den Calling Conventions von C werden Parameter auf dem Stack übergeben.
- Ablauf:
  - Der Aufrufer legt die Parameter auf den Stack (`push`).
  - Der Aufrufer ruft die Funktion auf (`call`).
  - Die Funktion wird bearbeitet.
  - Die Funktion kehrt mit `ret` zurück.
  - Der Aufrufer nimmt die Parameter vom Stack (`add esp,wert`).
- Achtung: Parameter bleiben auf dem Stack liegen.
  - Der Aufrufer muss aufräumen!

Beispiel: Parameterübergabe I

```
      push dword 0x10   ; Parameter auf den Stack legen
      push dword 0x20
      dump_stack2 1,0,2 ; Stack ausgeben
      call max          ; Funktion aufrufen
      dump_regs 1       ; Registerwerte ausgeben
      add esp, 8        ; Parameter vom Stack nehmen
      ...
max:  mov ebx, [esp+8]  ; Achtung Rücksprungadr. steht auf dem Stack
      cmp ebx, [esp+4]
      jl bkl
      mov eax, ebx      ; Rückgabe über eax
      jmp ende
bkl:  mov eax, [esp+4]
ende: ret
```

Beispiel: Parameterübergabe II

Ausgabe:

```
Stack Dump # 1
EBP = BFFFDF58 ESP = BFFFDF30
  +8  BFFFDF38  BFFFDFB4
  +4  BFFFDF34  00000010
  +0  BFFFDF30  00000020
```

Beispiel: Parameterübergabe III

```
      push dword 0x10    ; Parameter auf den Stack legen
      push dword 0x20
      dump_stack2 1,0,2  ; Stack ausgeben
      call max           ; Funktion aufrufen
      dump_regs 1        ; Registerwerte ausgeben
      add esp, 8         ; Parameter vom Stack nehmen
      ...
max:  mov ebx, [esp+8]   ; Achtung Rücksprungadr. steht auf dem Stack
      cmp ebx, [esp+4]
      jl bkl
      mov eax, ebx       ; Rückgabe über eax
      jmp ende
bkl:  mov eax, [esp+4]
ende: ret
```

Beispiel: Parameterübergabe IV

Ausgabe:

```
Register Dump # 1
EAX = 00000020 EBX = 00000010 ECX = 00000001 EDX = 401570C0
ESI = 40014020 EDI = BFFFDFB4 EBP = BFFFDF58 ESP = BFFFDF30
EIP = 08048418 FLAGS = 0287       SF       PF CF
```

Lokale Variablen

- Lokale Variablen werden ebenfalls auf dem Stack abgelegt.
- Wenn sich ESP innerhalb einer Funktion ändert, wird es schwierig auf die Parameter und die lokalen Variablen zuzugreifen.
- daher: ein weiteres spezielles Register
  - Extended Base Pointer (EBP)
  - EBP markiert den Beginn einer Funktion auf dem Stack.
  - Bei Funktionseintritt wird der alte Wert von EBP auf den Stack gelegt (`push`),
    - ... um ihn für die Rückkehr zur aufrufenden Funktion zu speichern.
  - Dann wird EBP der Wert von ESP zugewiesen.
  - Dann wird Platz für lokale Variablen geschaffen:
    - Addiere zu ESP die Größe der lokalen Variablen.
  - Bei Funktionsende werden die lokalen Variablen vom Stack gelöscht und der alte Wert von EBP wieder hergestellt.

Vereinfachung

- Das Setzen von EBP und das Reservieren von Speicher für lokale Variablen nennt man Prolog einer Funktion.
- Das Löschen der Variablen vom Stack und das Wiederherstellen von EBP nennt man Epilog einer Funktion.
- Da Prolog und Epilog sehr häufig vorkommen, gibt es spezielle Instruktionen dafür:
  - `enter op1, op2`
    - `push ebp`
    - `mov ebp, esp`
    - `sub esp op1`
    - op2 wird bei den C Calling Conventions nicht benötigt
  - `leave`
    - `mov esp, ebp`
    - `pop ebp`
- Achtung: `enter` benötigt mehr Zeit für die Ausführung, als die einzelnen Instruktionen zusammen!

Aufruf einer C-Funktion

- C-Funktionen können direkt von Assembler aus aufgerufen werden.
- Parameter werden in umgekehrter Reihenfolge auf den Stack gelegt (der letzte Parameter als erstes):
  - sinnvoll, da bei manchen Funktionen in C die Anzahl der Parameter variabel ist;
  - Einer der nicht variablen Parameter bestimmt, wie viele Parameter vorhanden sind.
  - Die nicht variablen Parameter stehen damit relativ zum EBP an einer festen Stelle.
  - Beispiel: `printf("x = %d\n",100);`
- Unter Windows: `"_"` dem Funktionsnamen voranstellen
- Rückgabe: erfolgt bei Integer-Werten über eax

Beispiel: Aufruf von `printf`

```
extern printf ; Versprechen, dass das Symbol printf später hinzukommt

segment .data
x:      dd 100
format: db "X = %d",10,0  ; 10=lf, 0=Ende des Strings (C Konvention)

segment .text
        global  asm_main
asm_main:
        enter   0,0       ; Funktion initialisieren
        pusha

        push dword [x]    ; x auf den Stack legen
        push dword format ; die Adresse (!) von format auf den Stack legen
        call printf       ; Funktion aufrufen
        add esp, 8

        popa              ; zu C zurückkehren
        mov     eax, 0
        leave
        ret
```

Retten von Registern

- C erwartet, dass einige Register nach Rückkehr aus einer Funktion wieder ihren alten Wert beinhalten.
- Unterstützt durch `pusha` und `popa`:
  - `pusha` schreibt die Register EAX, ECX, EDX, EBX, ESP, EBP, ESI und EDI auf den Stack (=32 Byte) und reduziert dann ESP um 32.
  - `popa` holt die Register in umgekehrter Reihenfolge vom Stack und erhöht ESP dann um 32.

Beispiel: Retten von Registern

```
%include "asm_io.inc"

segment .text
        global  asm_main
asm_main:
        enter   0,0             ; Funktionseintritt
        pusha

        mov     ebx, 1          ; lade eine 1 in das Register ebx
        dump_regs 1             ; gib Register aus

        popa                    ; nach C zurückkehren
        mov     eax, 0          ; Rückgabeparameter setzen
        leave
        ret
```

Übergabe von Zeigern

- In C werden häufig Zeiger auf lokale Variablen als Parameter übergeben:

```
#include<stdio.h>

void red(int* x) {
  *x=*x - 5;
}

int main(int argc, char* argv[]) {
  int x=10;
  red(&x);
  printf("Inhalt von x : %d\n",x);
  return 0;
}
```

- In Assembler bedeutet das:
  - Übergebe die Adresse (auf dem Stack), an der x steht.

Wie berechnet man Adressen?

- Wie kann man die Adresse von x berechnen, um sie dann als Parameter auf den Stack zu legen?
- mühsam:
  - EBP in EAX laden
  - dann davon den Offset zum Speicherplatz von x abziehen
- einfacher: Adressberechnung mit lea (load effective address)
- `lea op1, op2`
  - `op2` muss eine Speicheradresse sein z.B. `[ebp-8]`
  - dann wird die Speicheradresse berechnet und das Ergebnis in `op1` gespeichert
  - es wird NICHT der Inhalt dieser Speicheradresse in `op1` gespeichert

Beispiel: load effective adress

```
    mov dword [ebp-4], 10
    lea eax, [ebp-4]    ; Speicheradresse von x bestimmen
    push eax            ; Speicheradresse von x auf den Stack legen
    call red
    add esp, 4
    dump_stack 1,1,0    ; Stack ausgeben (hier relativ zu ebp)

red:enter 0,0
    mov eax, [ebp+8]
    sub dword [eax], 5
    leave
    ret
```

Ausgabe:

```
Stack Dump # 1
EBP = BFFFE4D8 ESP = BFFFE4B4
  +0  BFFFE4D8  BFFFE4E8
  -4  BFFFE4D4  00000005
```

Stack Frame

- Mit Stack Frame bezeichnet man den Bereich auf dem Stack, der zu einer Funktion gehört.
- Das ist der Bereich von EBP bis zum ESP:
  - alter EBP
  - gesicherte Register-Inhalte
  - lokale Variablen
  - sonstiges
- Achtung: Übergebene Parameter und Rücksprungadresse gehören daher zum Stackframe der aufrufenden Funktion.
  - darüber kann man sich streiten . . .

Von C zu Assembler

- Ein Compiler übersetzt Programme einer Hochsprache in Instruktionen des Befehlssatzes eines Prozessors.
- Das kann man mit gcc sehr gut ausprobieren.
- Was macht gcc, wenn er für eine .c Datei aufgerufen wird:
  - zuächst wird der C-Präprozessor (cpp) ausgeführt
    - Resultat: eine .C Datei
  - dann wird kompiliert: das C-Programm wird in ein Assembler-Programm übersetzt
    - Resultat: eine .s Datei (leider nicht in NASM-, sondern in GAS-Syntax)
  - Dann wird assembliert (mit gas): das Assembler-Programm wird in Instruktionen des Prozessors umgesetzt.
    - Resultat: eine .o Datei
  - Dann werden alle benötigten Dateien und Bibliotheken zusammengelinkt (mit ld).
    - Resultat: ausführbares Programm

Beispiel: Von C zu Assembler

```c
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  return 0;
}
```

- Übersetzen mit `gcc hello.c -S`
- Ergebnis: *hello.s*

Assembler-Code

```
.file  "hello.c"
   .section   .rodata
.LC0:
   .string    "Hello World!\n"
   .text
.globl main
   .type      main, @function
main:
   pushl  %ebp            # push ebp
   movl   %esp, %ebp      # mov ebp, esp
   subl   $8, %esp        # sub esp, 8
   andl   $-16, %esp      # and esp, -16 (löscht die unteren 4 Bit)
   movl   $0, %eax        # mov eax, 0
   subl   %eax, %esp      # sub esp, eax
   movl   $.LC0, (%esp)   # mov [esp], .LC0
   call   printf          # call printf
   movl   $0, %eax        # mov eax, 0
   leave                  # leave
   ret                    # ret
   .size     main, .-main
   .section  .note.GNU-stack,"",@progbits
   .ident    "GCC: (GNU) 3.3.2 20040119 (Gentoo Linux 3.3.2-r7)"
```

Handoptimierte Version

```
.file  "hello.c"
   .section   .rodata
.LC0:
   .string    "Hello World!\n"
   .text
.globl main
   .type      main, @function
main:
   pushl  %ebp             # push ebp
   movl   %esp, %ebp       # mov ebp, esp
   subl   $4, %esp         # sub esp, 4
   movl   $.LC0, (%esp)    # mov [esp], .LCO
   call   printf           # call printf
   movl   $0, %eax         # mov eax, 0
   leave                   # leave
   ret                     # ret
```

Unfair! Optimierung automatisch mit `gcc -O<Grad> hello.c`.
Beispiel: `gcc -O3 hello.c`

Disassembler

- Ein Disassembler kann die Maschineninstruktionen in einem Programm in Assembleranweisungen „zurückübersetzen“.
- Dabei geht normalerweise einiges verloren: Labels und Kommentare sind im Maschinencode nicht mehr enthalten.
- Aber der Weg über den in den Debugger gdb eingebauten Disassembler kann interessant sein:
  - wenn mit der Kommandozeilenoption -ggdb kompiliert wurde...
  - ...dann kann gdb beispielsweise gezielt einzelne Funktionen des fertig übersetzten Programms disassemblieren.
  - Hierfür zunächst `gdb programmname` aufrufen...
  - ...dann mit `disas funktionsname` eine Funktion disassemblieren.
  - gdb kann mit dem Befehl `quit` beendet werden.
- (Auch sonst lohnt sich ein Blick auf gdb oder andere Debugger – das kann ein sehr wertvolles Werkzeug sein!)

Disassemblieren mit gdb

```
$ gcc -ggdb -o test test.c
$ gdb test
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
(gdb) disas main
Dump of assembler code for function main:
0x080483c4 <main+0>:    lea    0x4(%esp),%ecx
0x080483c8 <main+4>:    and    $0xfffffff0,%esp
0x080483cb <main+7>:    pushl  -0x4(%ecx)
0x080483ce <main+10>:   push   %ebp
0x080483cf <main+11>:   mov    %esp,%ebp
0x080483d1 <main+13>:   push   %ecx
0x080483d2 <main+14>:   sub    $0x4,%esp
0x080483d5 <main+17>:   movl   $0x80484b0,(%esp)
0x080483dc <main+24>:   call   0x80482f4 <puts@plt>
0x080483e1 <main+29>:   mov    $0x0,%eax
0x080483e6 <main+34>:   add    $0x4,%esp
0x080483e9 <main+37>:   pop    %ecx
0x080483ea <main+38>:   pop    %ebp
0x080483eb <main+39>:   lea    -0x4(%ecx),%esp
0x080483ee <main+42>:   ret
End of assembler dump.
(gdb) quit
$
```

Buffer Overflow Exploits

Hacken auf dem Stack

Exploit (engl. to exploit = ausnutzen):

- Programm oder Daten, die eine Sicherheitslücke in anderen Computerprogrammen ausnutzen, um die Funktion zu stören oder sich illegitim Privilegien zu verschaffen.

Exploits mittels Pufferüberläufen I

- Eine sehr bekannte Form von Exploits nutzt so genannte Pufferüberläufe auf dem Stack aus. Grundprinzip:
  - Ein Programm nimmt Eingaben entgegen (von der Tastatur, aus einer Datei, aus dem Netzwerk, . . .).
  - Die Eingabe wird in eine lokale Variable einer Funktion kopiert, die eine festgelegte Größe hat.
  - Die Eingabe ist länger als diese Größe.
  - Der Programmierer hat vergessen, dies zu überprüfen.
  - Damit werden die vom Benutzer gelieferten Daten über das Ende des dafür reservierten Speicherbereiches hinaus geschrieben und überschreiben andere Daten. . .
  - . . . insbesondere die Rücksprungadresse auf dem Stack!
  - Beim Zurückkehren aus der aktuellen Funktion wird so an die “falsche“ Adresse zurückgesprungen.
  - Wenn der Angreifer geschickt vorgeht, kann er die Rücksprungadresse beliebig wählen!

Exploits mittels Pufferüberläufen II

- Im Prinzip also ein ganz einfaches Vorgehen:
  - Bestimme, an welche Adresse gesprungen werden soll.
  - Bestimme, wie groß der Puffer ist, in den geschrieben wird.
  - Bestimme, wie weit hinter dem Ende des Puffers die Rücksprungadresse liegt.
  - Baue eine entsprechende Eingabe.
  - „Füttere“ sie in das angegriffene Programm.
- Durch Pufferüberläufe ausnutzbare Schwachstellen waren (und sind) sehr häufig.
- Moderne Compiler und Betriebssysteme enthalten zunehmend ausgefeilte Schutzmechanismen dagegen – das heißt aber nicht, dass man als Programmierer/in nicht aufpassen muss!

Vertiefungsübung

Was?	Sicherheitslücken: Von Gespenster in Prozessoren
Wann?	Donnerstag, 08:30 Uhr
Wo?	2522.U1.55