Maschinencode Dateiformat und Stackframes

Proseminar C-Programmierung - Grundlagen und Konzepte

Julian M. Kunkel

julian.martin.kunkel@informatik.uni-hamburg.de

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

27-05-2011

Motivation

Hintergründe verstehen für

- Fehleranalyse
- Hacking
- Code-Optimierung
- Code von verschiedenen Sprachen gemeinesam nutzen

1. Dateiformat

2. Stack

3. Backtrace

4. Hacking

1 Dateiformat

2 Stack

3 Backtrace

4 Hacking

Fragen

- Woher weiß der Linker wie Objektdateien zusammen gehören?
- Wie werden Programme in den Speicher geladen?

Executable and Linking Format (ELF)

Dateiarten

- Executable: Ausführbare Datei
- Relocatable: Objekt Datei
- Shared object: Bibliothek

ELF Beschreibt Inhalte der Binärdateien

- Zwei Sichten: Programmausführung vs. Compilierung
- Programmkopf-Tabelle zeigt auf “Segmente”
- “Sektionen” enthalten informationen für Linker und Relocation

DWARF bettet Debugging Informationen in ELF ein...

ELF Aufbau

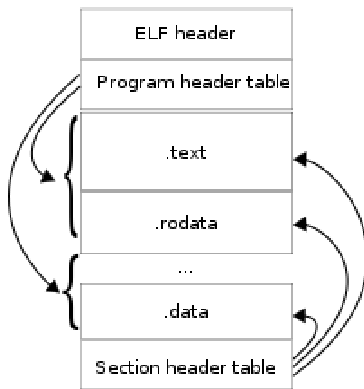

Figure: ELF file structure[1]

[1] Surueña, "Layout of an ELF file", http://en.wikipedia.org/wiki/Executable_and_Linkable_Format, 23.05.2011

Wichtigste Bestandteile der Objektdateien

- “(RO)Data”: Datensegment, enthält initalisierte globale und statische Variablen
- “BSS”: Block Started by Symbol, 0-initalisierte statische Variablen
- “Text”: Enthält ausführbaren Maschinencode
- Symboltabelle: Symbolnamen für den Linker relevant
- “Dynsym”: Enthält globale Symbole ist zur Laufzeit bekannt

Frage an das Publikum

Warum gibt es Dynsm und eine “normale” Symboltabelle?

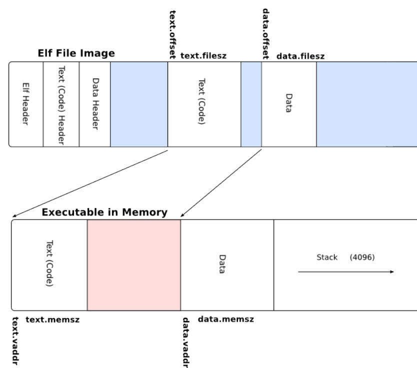

Figure: Executing an ELF file[2]

[2] Jhawthorn, "Executable image and elf binary can being mapped onto each other", http://wiki.osdev.org/ELF, 23.05.2011

Werkzeuge

- `nm` – Symbole von Objektdateien anzeigen
- `objdump` – Objekt-Dateien analysieren
  - -d – Code disassemblieren
  - -s – Inhalt der Sektionen
  - -t – Symboltabelle
  - -T – Dynamische Symboltabelle
- Symbole zur dynamischen Symboltabelle hinzufügen mit: `gcc -rdynamic`
- Symboltabelle und andere (überflüssigen) Bereiche entfernen: `strip`
- Assembler erzeugen: `gcc -S <DATEI.c>`

Dynamische Symbole

Bibliotheken

- Shared-libraries stellen Dsym zur Verfügung
- Programm-Loader lädt Shared-Libaries

Manuelle Verwendung

- Zur Laufzeit Code nachladen
  - Module programmieren
- `dlopen()`, `dlsym()`, `dlclose()`

Demo: stackframe-simple.c

1 Dateiformat

2 Stack

3 Backtrace

4 Hacking

Stapelspeicher

- LIFO Speicherbereich
- Schnellere/Leichtere Verwaltung als Heap
- Jeder Thread bekommt einen Stack zugewiesen
- Stack wächst Richtung niedrigerer Adresse
- Funktionsaufrufe als Stack-Frame auf Stack speichern

Stack-Frame

Adressierung von Variablen

- Addressierung relativ zu dem Start des Frames
- “Base-Pointer” – zeigt auf Start des Frames
- “Stack-Pointer” – zeigt auf Ende des Stacks

Bei Funktionsaufruf

- Alten IP und BP merken
- Lokale Variablen auf Stack reservieren
- Registerzustand speichern
- BP, IP und SP umsetzen
- $\Rightarrow$ Zeitaufwand

Stack-Frame

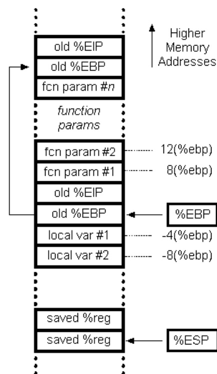

Figure: Stack-Frame[3]

[3] Steve Friedl, "", `http://www.unixwiz.net/techtips/win32-callconv-asm.html`, 23.05.2011

Aufrufkonventionen

Varianten

- **cdecl**: Nutzung des Stacks für Parameter und Rückgabewerte
  - Variablen Parameteranzahl ⇒ Aufrufer bereinigt Stack
- **stdcall**: Feste Anzahl an Parametern für eine Funktion
  - Funktion entfernt Parameter von Stack selbst

Optimierungen

- **Register**: Parameter werden in Registern abgelegt
- **Frame Pointer Omission**: Verwende SP zur Adressierung

Verschiedenes

Stackspeicher nutzen

- `alloca` reserviert Speicher auf Stack
- Vorsicht: Nach Stackabbau ungültig
- Vorsicht: Größe des Stacks ist limitiert

Stackgröße

- Vom System oder zur Kompilezeit eingestellt
- Unter Linux mittels `ulimit -a` anzeigen oder verändern

1 Dateiformat

2 Stack

3 Backtrace

4 Hacking

Backtrace

- Ein Backtrace ermittelt die IP’s der aufrufenden Funktionen
- Name der Funktion in welcher der IP gerade steht?

- GNU C Library (libc) stellt Backtraces zur Verfügung
- Funktionsnamen in dynamischer Symboltabelle benötigt

Demo: backtrace.c mit objdump und -rdynamic

1 Dateiformat

2 Stack

3 Backtrace

4 Hacking

Hacking der Stack-Frames

- Gespeicherten Rücksprungadresse auf Stack umsetzen
  - $\Rightarrow$ Bei Verlassen der Funktion wird anderer Code ausgeführt
- Basis von Buffer-Overflow Angriffen
  - $\Rightarrow$ Lokale Variable überschreibt gespeicherten IP

Stack-Frame hacken

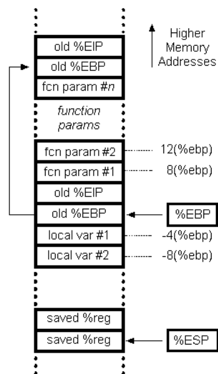

Figure: Stack-Frame[4]

[4] Steve Friedl, "", `http://www.unixwiz.net/techtips/win32-callconv-asm.html`, 23.05.2011

Demo: hacked.c

Zusammenfassung

- Compilezeit vs. Laufzeit
- ELF: Code, Datenbereiche, Symbolltabelle(n)
- ELF Dateien mit `objdump` analysieren
- Stack-Frames speichern Funktionsaufrufe
- Backtraces verstehen $\Rightarrow$ Symboltabelle und Stack-Frames!

Literatur

AMD.
Amd64 instruction-level debugging with sun studio dbx.

A. Bahrami.
Inside elf symbol tables.

S. Friedl.
Intel x86 function-call convetions – assembly view.

G. L. Manual.
Backtraces.

A. Müller.
Der stack frame.

OSDev.
Elf.

Phillip.
Using assembly language in linux.

Wikipedia.
Call stack.