

Frickeln

Sven Gregori, CN8

Phil Sutter, CN8

UnFUG SS 2008
Hochschule Furtwangen

10. April 2008

Binärtools

strings

- listet ASCII-Zeichenketten in Binärdateien
- Suchen einer Fehlermeldung in mehreren Programmen
- `strings /dev/mem` findet manchmal Passwörter

hexdump und bvi

- hexdump printet Daten in Hex-Format
- bvi „binary VI“

nm und objdump

- Tools zur Analyse von Binärdateien
- nm funktionale Untermenge von objdump

```
nm -n vmlinux >/tmp/syms && vimdiff /tmp/syms System.map
```

- objdump -h <file> gibt Section Header aus
- objdump -t <file> printet Symbole
- objdump -d <file> Disassembler

ctags

ctags

- „Indexierung von Codeverzeichnissen“
- erstellt Datei mit Tags zur Suche nach Definitionen/Deklarationen
- `ctags *` erstellt Datei `tags` im CWD über alle Dateien im Verzeichnis

ctags und vim

- braucht *tags*-Datei im CWD
- CTRL-5 und CTRL-T
- :tags zeigt Tag-Stack an
- :ts[elect] zeigt Match-List für aktuellen Tag, oder Regexp
- :help tags-and-searches

cscope

cscope

- textbasierter Source Code Browser
- Default auf *.c *.h *.l *.y
- kann aber auch C++ und Java Dateien verstehen
- hat Cross Referenz und Suchfunktion für
 - alle Referenzen von einem Symbol
 - globale Definitionen
 - Funktionen die andere aufrufen oder von anderen aufgerufen werden
 - Text Strings und Regular Expressions
 - Dateien und Dateien die andere Includen
- ncurses Frontend
- Backend im Hintergrund inklusive VIM Support \o/

cscope

- cscope generiert Datenbank und ruft Frontend auf
- cscope -b generiert nur Datenbank
- cscope -d öffnet Frontend ohne Datenbank Regenerierung
- cscope -b -q generiert zusätzlich "inverted index"
→ schnellere Suchzugriffe
- direkter Aufruf von $\${EDITOR}$ aus Frontend
- VIM greift auf cscope Datenbank und Funktionalität zu

VIM Kleinigkeiten

VIM

MiniBufExplorer Plugin

- Extra Fenster, das alle geöffneten Dateien anzeigt
- auswählen einer Datei zeigt sie an
- http://www.vim.org/scripts/script.php?script_id=159

TagList Plugin

- Source Code Browser
- listet globale Variablen, Funktionen, Macros, ...
- <http://vim-taglist.sourceforge.net/>

vimdiff

- Teil von VIM
- öffnet 2 oder mehr Dateien nebeneinander
- zeigt farblich Unterschiede in den Dateien an

GCC

GCC Warnings

- Zeigt Programmkonstrukte an, die früher oder später zu Problemem führen könnten
- `-w` unterdrückt alle Warnings
→ EVIL! Warnings sind wichtig!
- `-Wall` enabled einen Großteil der wichtigsten Warnungen
- `-Wextra` zusätzlich noch weitere spezielle Warnungen
- `-std=...` definiert zu verwendenden C Standard
- `-pedantic` erzwingt strikte Einhaltung des Standards item
`-pedantic-errors` behandelt pedantic Warnings als Fehler
- `-Werror` behandelt ALLE Warnungen als Fehler

→ soviel Warnings wie möglich anzeigen lassen und den Code so fixen, dass sie verschwinden

GCC `__attribute__(())`

- GCC Erweiterung
- ermöglicht besondere Eigenschaften für Variablen und Funktionen zu deklarieren
- Liste enthält ein oder mehrere, durch Komma getrennte Attribute
- bei Funktionen nur bei der Funktionsdeklaration möglich

```
int foo(void) __attribute__((noreturn));
```

```
...
```

```
int foo(void) { ... }
```

- bei Variablen muss es vor einer Initialwertzuweisung stehen

```
int foo __attribute__((unused)) = 23;
```

GCC `__attribute__(())`

`__attribute__((unused))`

- sagt dem Compiler, dass eine deklarierte Variable oder statische Funktion nicht verwendet wird
- verhindert Warnung darüber

`__attribute__((deprecated))`

- deklariert eine Variable oder Funktion als deprecated
- Compiler erzeugt Warnung, wenn die Variable oder Funktion verwendet wird

GCC `__attribute__(())`

`__attribute__((section))`

- teilt dem Compiler mit, die Variable oder Funktion in der definierten section abzulegen
- default normalerweise `.text` oder `.bss` (static)

`__attribute__((weak))`

- definiert weak symbol statt global Symbol
- Linker meckert nicht über nicht auflösbare weak Symbols
- nicht vorhandene Libraries können so zur Laufzeit ggf. ersetzt oder behandelt werden (Funktionseinschränkung)

GCC `__attribute__(())`

`__attribute__((aligned(x)))`

- Variablen werden auf x Bytes aligned
- x ist minimal-Angabe

`__attribute__((aligned))`

- Compiler sucht ideales Alignment aus

`__attribute__((packed))`

- kleinstmöglichstes Alignment wird versucht
- z.B. bei structs, wenn möglichst wenig Leerräume auftreten sollen - bei enum wird kleinstmöglicher Datentyp genommen

GCC `__attribute__(())`

`__attribute__((cleanup(fptr)))`

- definiert eine Callback Funktion die aufgerufen wird, wenn die zugehörige Variable ihre Sichtbarkeit verliert
- nur bei lokalen (auto) Variablen möglich
- `fptr` ist Pointer auf eine Funktion der Form
`void func(<type> *foo)`
- `foo` Parameter ist hierbei Pointer auf die Variable selbst
- `<type>` muss gleich wie der Variablentyp selbst sein
- Rückgabewerte der Callback Funktion werden ignoriert

GCC `__attribute__(())`

`__attribute__((alias(''target'')))`

- definiert ein Alias für die Funktion "target"
- in Kombination mit `weak` möglich

`__attribute__((always_inline))`

- weist den Compiler an, die entsprechende Funktion immer als inline Funktion zu behandeln, nicht nur wenn optimiert werden soll

GCC `__attribute__(())`

`__attribute__((constructor))`

- platziert Funktion in `.ctors` Section
- wird vor `main()` ausgeführt

`__attribute__((destructor))`

- platziert Funktion in `.dtors` Section
- wird nach `main()` ausgeführt

GCC `__attribute__(())`

`__attribute__((fastcall))`

- die ersten zwei Funktionsparameter werden anstatt auf dem Stack in den Registern ECX und EDX abgelegt
- weitere Parameter wie gewohnt auf dem Stack
- nur x86

`__attribute__((regparm(x)))`

- bis zu x Funktionsparameter werden in den Registern EAX, ECX und EDX abgelegt
- gilt nicht für variable Parameterlisten
- nur x86, und nicht ganz problemlos

GCC `__attribute__(())`

`__attribute__((noreturn))`

- definiert, dass eine Funktion nie zurückkehren wird
- z.B. `abort()` oder `exit()`
- Compiler erzeugt Warning, wenn sie es doch tut

`__attribute__((used))`

- definiert, dass eine Funktion benutzt wird, auch wenn es nicht den anschein hat
- z.B. wenn sie nur in inline Assembler benutzt wird

lint

lint

- statische Code-Analyse
- Linux-Implementierung: `splint` (kann mehr)

Makefiles und Autotools

Makefiles

- Zeilen können sein:
 - Kommentare
 - Variablenzuweisungen
 - Target-Definitionen
- Targets bestehen aus:
 - Namen
 - evtl. Dependencies
 - evtl. Liste von Kommandos
- Abarbeitung:
 - Target suchen
 - Dependencies (als Targets) suchen
 - Befehle Abarbeiten (Abbruch bei Fehler)

Autotools

- `configure` generiert *Makefile* aus *Makefile.in*
- `autoconf` generiert *configure* aus *configure.ac*
- `automake` generiert *Makefile.in* aus *Makefile.am*

valgrind

valgrind

- Laufzeitanalyse und Debugging von Programmen
- verschiedene Analyse-Funktionalitäten, default ist *memcheck*

gprof

gprof

- zeigt Profiling Informationen und Call Graph an
- eventuelle Bottlenecks können so erkannt werden
- Programm muss dafür mit `gcc -pg` kompiliert werden
- Ausführen von Programm erzeugt Datei `gmon.out` auf die `gprof` dann zugreift
- Ablauf:

```
$ ls
foo
$ ./foo
$ ls
foo gmon.out
$ gprof ./foo
```

gprof

- Auflistung wie oft eine Funktion aufgerufen wurde (auch auf Source Code Ebene)
- Zeit die dafür benötigt wurde
- welche Funktion was für Funktionen aufgerufen hat (Call Graph)

Alternative ist z.B. *oprofile*

- klemmt sich in den Kernel rein
- daher kein `gcc -pg` nötig
- ..aber noch nicht mit herumgespielt ;/

strace & ltrace

strace

- Monitoring von System Calls und Signalen
- benutzt *ptrace* Feature im Linux Kernel
- ermöglicht
 - Monitoring von System Calls inklusive deren Parametern und Rückgabewerte
 - Anzeigen von empfangenen Signalen
 - Statistik über aufgerufene System Calls (Anzahl und Zeit)
 - einhängen in einen bereits laufenden Prozess
 - Filter auf bestimmte Events
 - ...
- Ausgabe in Datei möglich - VIM kennt `filetype=strace ;`

ltrace

- ähnlich wie `strace`, auch was Parameter angeht
- primär jedoch Monitoring von Library Calls
- Monitoring von System Calls aber auch möglich - zusätzlich oder exklusiv
- funktioniert aber nur mit dynamisch gelinkten Programmen

Debugging mit ld.so

ld.so - the Linux loader

- lädt Shared Libraries zur Laufzeit
- kann Debug Informationen darüber erzeugen und ausgeben
- hat Umgebungsvariablen LD_DEBUG und LD_DEBUG_OUTPUT
- LD_DEBUG setzt Debug Optionen
- LD_DEBUG_OUTPUT definiert optional Dateiname für Ausgabe (default stdout)
- LD_DEBUG=help <programmname>
listet mögliche Optionen auf
- LD_DEBUG=<option> <programmname>
erzeugt Debug Informationen

GDB

GDB - The GNU Debugger

- Programmausführung innerhalb des Debuggers ermöglicht
 - Ausführung an definierten Stellen unterbrechen (Breakpoints)
 - untersuchen was passiert ist
 - Variablenwerte auslesen und ändern
- `gcc -g` erzeugt Debugging Symbole in der Binärdatei
- ermöglicht Debugging auch auf Source Code Level
- optional kann core dump mitgegeben werden
- mit *gdbserver* auch remote Debugging über serielle Konsole oder TCP möglich

GDB - Befehle (Auszug)

- `list [[<file>:]<function>|<linenum>] |*<address>]`
- `info [all-]registers`
- `info variables [<pattern>]`
- `print <variable> [= <value>]`
- `info address <symbol>`
- `info symbol <address>`
- `info source[s]`
- `info files`
- `info frame`
- `[info stack|backtrace|bt]`
- `info locals`
- `info proc`

GDB - Befehle (Auszug)

- `disassemble [<start address> [<end address>]]`
- `run [<parameters>]`
- `attach <pid>`
- `target remote [<serial port>| [<host>]:<port>]`
- `signal <signal>`
- `next[i]`
- `step[i]`
- `jump [<line>|*<address>]`

- `break [<line>|<function>|*<address> [if <condition>]]`
- `watch <condition>`

- `help [<topic>]`

Erwähnenswertes

Erwähnenwertes

lxr

- HTML Source Code Cross Reference
- Source Code Browser mit links auf sämtliche Funktionen, Variablen, Macros etc.

doxygen

- Code Kommentieren kann sinnvoll sein ;)
- kompatibel zu Javadoc
- unterstützt C, C++, Java, Ada, VHDL, Python, PHP, C#, ...
- generiert HTML, \LaTeX , Man Pages, chm, ...

Links

Links

cscope

<http://cscope.sourceforge.net/>

ctags

VIM: :help tags-and-searches

GCC

<http://gcc.gnu.org/onlinedocs/> :)

make

<http://www.gnu.org/software/make/manual/make.html>

Links

valgrind

<http://valgrind.org/>

gprof

<http://sourceware.org/binutils/docs/gprof/index.html>

Debugging Allgemein

<http://www.informit.com/articles/article.aspx?p=377306&seqNum=2>

GDB

<http://www.gnu.org/software/gdb/>

<http://www.oreilly.de/german/freebooks/rlinux3ger/ch142.html>

Fragen?