

Reverse Engineering (RE)

Team 3
Jan Gossens & Martin Morawetz

16.05.2017

- 1 Einführung
- 2 Vorgehensweisen von Reverse Engineering
 - Statische Analyse
 - Dynamische Analyse
 - Decompiler
- 3 Live Demo
- 4 Abwehr von Reverse Engineering
- 5 Tools

Was ist Reverse Engineering?

- Rekonstruktion des Zustandes vor dem Kompilieren
- Binary zu Assembler oder Source Code

Anwendung von RE

- Analyse von Malware
- Verlust von Source Code
- Fehlersuche
- Neugier
- Sicherheitsanalyse
- Entschlüsselung von Ransomware

Begriffserklärungen

Disassembler vs. Decompiler

- Disassembler, Maschinen- zu Assembler-Code
- Decompiler, Maschinen- zu Source oder Pseudo-Code

Begriffserklärungen

Statisch vs. Dynamisch

- Statische Analyse, Analyse ohne Ausführung
 - sehr allgemein
 - Register- / Speicherwerte zur Laufzeit nicht vorhanden
 - Tools: objdump, strings
- Dynamische Analyse, Analyse mit / während Ausführung
 - Analyse für speziellen Programmablauf
 - sehr detailliert, manipulierbar
 - oft nur auf VM möglich
 - Tools: Debugger, ltrace, strace

Strings

- Statische Methode zum Finden von ASCII Strings
- Durchsucht Binärdateien
- Nützlich um schnellen Überblick zu erhalten
- Zeigt nur Strings mit mindestens 4 Zeichen an
- Manchmal findet man so auch das gesuchte "Passwort"

Objdump

- Statischer Disassembler
- `objdump -d`: diassembled eine Binärdatei
- `objdump -f`: Zeigt Header an

```
example1:      file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x00000000000000680
```


ltrace & strace

ltrace

- Zeigt Library Calls zur Laufzeit an

strace

- Zeigt System Calls zur Laufzeit an
- Gut um Schadsoftware zu erkennen

gdb

- gdb - GNU Debugger
- Dynamische Programmanalyse und -veränderung
- Nützlich um Programmfehler zu finden und Schwachstellen zu finden
- Sehr hardwarenah

gdb - PEDA

- PEDA - Python Exploit Development Assistance for GDB
- Nützliche, schlanke Erweiterung für GDB
- Zeigt zu jeder Zeit Register und Stack an
- Zusätzliche Features und Befehle
 - zB ropgadget/ropsearch für ROP

gdb - PEDA

```

[-----registers-----]
RAX: 0x555555547f2 (<main>: push rbp)
RBX: 0x0
RCX: 0x0
RDX: 0x7fffffff128 --> 0x7fffffff433 ("PERL_LOCAL_LIB_ROOT=/home/martin/perl5:/home/martin/perl5")
RSI: 0x7fffffff118 --> 0x7fffffff3d9 ("/home/martin/Documents/GoogleDrive/SecurityLab2017/exercises/reverse_engineering/example1")
RDI: 0x1
RBP: 0x7fffffff030 --> 0x55555554880 (<_libc_csu_init>: push r15)
RSP: 0x7fffffff030 --> 0x55555554880 (<_libc_csu_init>: push r15)
RIP: 0x555555547f6 (<main+4>: lea rsp,[rsp-0x1040])
R8 : 0x555555548f0 (<_libc_csu_fini>: repz ret)
R9 : 0x7ffff7de9750 (<_dl_fini>: push rbp)
R10: 0x4
R11: 0x1
R12: 0x55555554680 (<_start>: xor ebp,ebp)
R13: 0x7fffffff110 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-----code-----]
0x555555547f1 <check_password+65>: ret
0x555555547f2 <main>: push rbp
0x555555547f3 <main+1>: mov rbp,rbp
=> 0x555555547f6 <main+4>: lea rsp,[rsp-0x1040]
0x555555547fe <main+12>: or QWORD PTR [rsp],0x0
0x55555554803 <main+17>: lea rsp,[rsp+0x1020]
0x5555555480b <main+25>: mov DWORD PTR [rbp-0x14],edi
0x5555555480e <main+28>: mov QWORD PTR [rbp-0x20],rsi
[-----stack-----]
0000| 0x7fffffff030 --> 0x55555554880 (<_libc_csu_init>: push r15)
0008| 0x7fffffff038 --> 0x7ffff7a56511 (<_libc_start_main+241>: mov edi,eax)
0016| 0x7fffffff040 --> 0x400000
0024| 0x7fffffff048 --> 0x7fffffff118 --> 0x7fffffff3d9 ("/home/martin/Documents/GoogleDrive/SecurityLab2017/exercises/reverse_engineering/example1")
0032| 0x7fffffff050 --> 0x1f7b9d888
0040| 0x7fffffff058 --> 0x555555547f2 (<main>: push rbp)
0048| 0x7fffffff060 --> 0x0
0056| 0x7fffffff068 --> 0x36f92d079a623451
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x0000555555547f6 in main ()
gdb-peda$

```

gdb - Wichtige Befehle Teil 1

- Binary starten: `run/r`
 - nimmt Argumente
- Breakpoints setzen: `break/b main/*0x5555555547fe`
 - entweder Funktionsname oder Pointer auf Speicheradresse
- Breakpoints auflisten: `info breakpoints/b`
- Breakpoints entfernen: `del <n>`
 - n ist Index des Breakpoints
- Step into: `si`
- Step over: `ni`
- Ausführen bis zum Ende der Funktion: `finish`
- Ausführen bis zum nächsten Breakpoint: `continue/c`

gdb - Wichtige Befehle Teil 2

- Register anzeigen lassen: info registers
 - mit PEDA nicht nötig
- Funktionen anzeigen lassen: info functions
- Disassemble: disassemble/disas main/*0x5555555547fe
- Vergangene Funktionsaufrufe: backtrace
- Strings anzeigen lassen: print (char*)0x5555555547fe
- Speicher verändern: set *0x7fffffff3e4 = <newValue>
 - Pointer auf Speicheradresse
 - 1 Byte wird per default geändert
 - n Byte per Array: set {char[N+1]} 0x.... = "StringOfLengthN"
- Section Adressen anzeigen: info file
 - Zeig auch Start Adresse(Entry Point) an

gdb - Wichtige Befehle Teil 3.1

- E'x'amine memory: x/<Anzahl><Art><Ausgabe>
- x/32wx \$esp: 32 words in Hexadezimal an Speicher auf den \$esp zeigt
- x/32wx *0x7fffffff3e4: 32 words in Hexadezimal an expliz. Speicher-Adresse
- Mögliche Ausgaben:
 - x - Integer in Hexadezimal
 - d - signed Integer
 - u - unsigned Integer
 - o - Integer in Oktal
 - t - Integer in Binär
 - a - Speicheradresse
 - f - Floating Point
 - c - Character Constant
 - i - Assembler/Maschineninstruktionen
 - s - Strings
 - Default ist 'x'

gdb - Wichtige Befehle Teil 3.2

- Mögliche Arten:
 - b - Bytes
 - h - halfwords (2 Byte)
 - w - words (4 Byte)
 - g - giant words (8 Byte)
- Bei Strings als Ausgabe ist die Art automatisch 'b'
- Bei Maschineninstruktionen der Ausgabe 'i' wird die Art ignoriert

IDA

- IDA - Interactive Disassembler
- Entwickelt von Hex-Rays
- Mächtigstes Tool für Reverse Engineering
- Kann aus Binärdateien C ähnlichen Pseudo Code rekonstruieren
- Üblicherweise nur selbe Semantik wie ursprünglicher Code
 - Kompilieren ist verlustbehaftet
 - fehlende Kommentare + Variablennamen
 - Optimierungen
- Sehr beliebt um Kopierschutz zu umgehen

IDA - Assembler View

The screenshot displays the IDA Pro interface with the Assembler View selected. The main window shows assembly code for the 'main' function, which is a C program that checks a password and prints a message. The code is as follows:

```

.text:00000000000007F0      leave
.text:00000000000007F1      retn
.text:00000000000007F1      check_password  endp
.text:00000000000007F2
.text:00000000000007F2      ; ===== SUBROUTINE =====
.text:00000000000007F2      ;
.text:00000000000007F2      ; Attributes: bp-based frame
.text:00000000000007F2      ;
.text:00000000000007F2      ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00000000000007F2      public main
.text:00000000000007F2      main          proc near          ; DATA XREF: _start+1df0
.text:00000000000007F2          = qword ptr -20h
.text:00000000000007F2      var_20
.text:00000000000007F2      = dword ptr -14h
.text:00000000000007F2      var_14
.text:00000000000007F2      = qword ptr -8
.text:00000000000007F2      var_8
.text:00000000000007F2      = qword ptr 0
.text:00000000000007F2      var_s0
.text:00000000000007F2
.text:00000000000007F2      push rbp
.text:00000000000007F2      mov rbp, rsp
.text:00000000000007F6      lea rsp, [rsp-1040h]
.text:00000000000007FE      or [rsp+var_s0], 0
.text:0000000000000803      lea rsp, [rsp+1020h]
.text:0000000000000808      mov [rbp+var_14], edi
.text:000000000000080E      mov [rbp+var_20], rsi
.text:0000000000000812      mov esi, 80h ; size
.text:0000000000000817      mov edi, 1 ; nmem
.text:000000000000081C      call _calloc
.text:0000000000000821      mov [rbp+var_8], rax
.text:0000000000000825      lea rdi, s ; "Welcome to our little password guessing"...
.text:000000000000082C      call puts
.text:0000000000000831      mov rax, [rbp+var_8]
.text:0000000000000835      mov rsi, rax
.text:0000000000000838      lea rdi, aS ; "%s"
.text:000000000000083F      mov eax, 0
.text:0000000000000844      call __isoc99_scanf

```

The left pane shows the Functions window with the following entries:

Function name	Segment
__init_proc	init
puts	plt
_calloc	plt
_strcmp	plt
__isoc99_scanf	plt
__cxa_finalize	plt.got
_start	text
deregister_tm_clones	text
register_tm_clones	text
_do_global_ctors_aux	text
frame_dummy	text
check_password	text
main	text
__libc_csu_init	text
__libc_csu_fini	text
_term_proc	fini
puts@G.LIBC_2_2_3	extern
__libc_start_main@G.LIBC_2_2_3	extern
calloc@G.LIBC_2_2_3	extern
strcmp@G.LIBC_2_2_3	extern
__isoc99_scanf@G.LIBC_2_7	extern
__cxa_finalize@G.LIBC_2_2_3	extern
puts	extern
__libc_start_main	extern
calloc	extern
strcmp	extern
__isoc99_scanf	extern
_imp_cxa_finalize	extern
_gmon_start_	extern
_Jv_RegisterClasses	extern

IDA - Hex View

The screenshot displays the IDA Pro Hex View window. The main pane shows assembly code for the `main` function, starting at address `00000000000000A2`. The code includes instructions like `push`, `call`, `strncpy`, `scanf`, `finalize`, `puts`, and `register_classes`. The hex view shows the corresponding byte sequences, and the pseudocode view shows the high-level logic of the program.

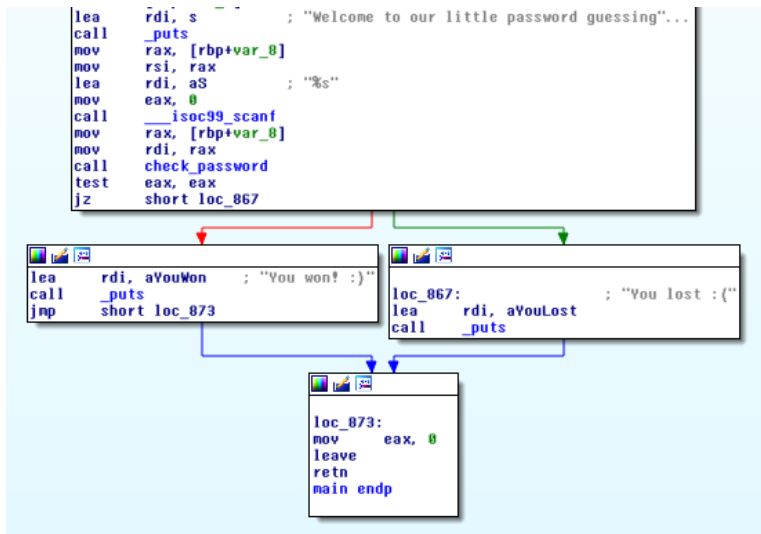
Function name	Segment	Address	Hex	Comment
<code>main</code>	<code>text</code>	<code>00000000000000A2</code>	<code>29 E5 48 83 EC 08 C1</code>	<code>FD 03 E8 4F FD FF FF</code>
<code>main</code>	<code>text</code>	<code>00000000000000A3</code>	<code>85 ED 74 20 31 D8 0F 1F</code>	<code>84 00 00 00 00 00 4C 89</code>
<code>main</code>	<code>text</code>	<code>00000000000000A4</code>	<code>3C 4D 89 F6 44 89 FF 41</code>	<code>FF 14 DC 48 83 C3 01 48</code>
<code>main</code>	<code>text</code>	<code>00000000000000A5</code>	<code>39 E0 75 EA 48 83 C4 08</code>	<code>5B 5D 41 5C 41 5D 41 5E</code>
<code>main</code>	<code>text</code>	<code>00000000000000A6</code>	<code>41 5F C3 90 66 2E 0F 1F</code>	<code>84 00 00 00 00 00 F3 C3</code>
<code>main</code>	<code>text</code>	<code>00000000000000A7</code>	<code>48 83 EC 08 48 83 C4 08</code>	<code>C3 01 00 02 00</code>
<code>main</code>	<code>text</code>	<code>00000000000000A8</code>	<code>00 00 00 00 62 6C 61 00</code>	<code>00 00 00 00 57 65 6C 63</code>
<code>main</code>	<code>text</code>	<code>00000000000000A9</code>	<code>6F 6D 65 20 74 6F 20 6F</code>	<code>75 72 20 6C 69 74 74 6C</code>
<code>main</code>	<code>text</code>	<code>00000000000000AA</code>	<code>65 20 70 61 73 73 77 6F</code>	<code>72 64 20 67 75 65 73 73</code>
<code>main</code>	<code>text</code>	<code>00000000000000AB</code>	<code>63 6E 67 20 67 61 6D 65</code>	<code>2E 20 47 69 76 65 20 69</code>
<code>main</code>	<code>text</code>	<code>00000000000000AC</code>	<code>74 20 61 20 74 72 79 3A</code>	<code>00 25 73 00 59 6F 75 20</code>
<code>main</code>	<code>text</code>	<code>00000000000000AD</code>	<code>77 6F 6E 21 20 3A 29 00</code>	<code>59 6F 75 20 6C 6F 73 74</code>
<code>main</code>	<code>text</code>	<code>00000000000000AE</code>	<code>20 2A 28 00 01 1B 03 30</code>	<code>44 00 00 00 07 00 00 00</code>
<code>main</code>	<code>text</code>	<code>00000000000000AF</code>	<code>08 FC FF FF 90 00 00 00</code>	<code>00 FD FF FF 00 00 00 00</code>
<code>main</code>	<code>text</code>	<code>00000000000000B0</code>	<code>18 FD FF FF 60 00 00 00</code>	<code>40 FE FF FF D0 00 00 00</code>
<code>main</code>	<code>text</code>	<code>00000000000000B1</code>	<code>0A FE FF FF F0 00 00 00</code>	<code>18 FF FF 1F 01 00 00 00</code>
<code>main</code>	<code>text</code>	<code>00000000000000B2</code>	<code>08 FF FF FF 58 01 00 00</code>	<code>14 00 00 00</code>
<code>main</code>	<code>text</code>	<code>00000000000000B3</code>	<code>00 00 00 00 01 7A 52 00</code>	<code>01 78 18 01 1B 0C 07 00</code>
<code>main</code>	<code>text</code>	<code>00000000000000B4</code>	<code>90 01 07 18 14 00 00 00</code>	<code>1C 00 00 00 00 FC FF FF</code>
<code>main</code>	<code>text</code>	<code>00000000000000B5</code>	<code>2B 00 00 00 00 00 00 00</code>	<code>00 00 00 00 14 00 00 00</code>
<code>main</code>	<code>text</code>	<code>00000000000000B6</code>	<code>00 00 00 00 01 7A 52 00</code>	<code>01 78 18 01 1B 0C 07 00</code>

IDA - Proximity View

The screenshot displays the IDA Pro interface in Proximity View. On the left, the 'Functions window' lists various functions and their segments. The main area shows a control flow graph (CFG) with nodes for 'start', 'main', and several other functions: 'calloc', 'strcpy', 'strcpy@GLIBC_2.2.5', '__strcpy_scan@GLIBC_2.7', '__strcpy_scan@GLIBC_2.2.5', '__strcpy_scan@GLIBC_2.2.5', 'check_password', 'aYouWon', and 'aYouLost'. Blue arrows indicate the flow of execution between these functions.

Function name	Segment
__init_proc	init
puts	git
__calloc	git
__strcpy	git
__strcpy_scan	DR
__strcpy_scan@GLIBC_2.2.5	DR@GL
__start	text
deregister_tm_stones	text
register_tm_stones	text
__do_global_ctors_aux	text
frame_dummy	text
check_password	text
main	text
__libc_csu_init	text
__libc_csu_fini	text
Item_proc	TR
puts@GLIBC_2.2.5	extern
__libc_start_main@GLIBC_2.2.5	extern
calloc@GLIBC_2.2.5	extern
strcpy@GLIBC_2.2.5	extern
__strcpy_scan@GLIBC_2.7	extern
__strcpy_scan@GLIBC_2.2.5	extern
__strcpy_scan@GLIBC_2.2.5	extern
__libc_start_main	extern
calloc	extern
strcpy	extern
__strcpy_scan	extern
__strcpy_scan@GLIBC_2.2.5	extern
__strcpy_scan@GLIBC_2.2.5	extern
__libc_start_main	extern
strcpy	extern
__strcpy_scan	extern
__strcpy_scan@GLIBC_2.2.5	extern
__strcpy_scan@GLIBC_2.2.5	extern
__libc_start_main	extern
strcpy	extern
__strcpy_scan	extern
__strcpy_scan@GLIBC_2.2.5	extern
__strcpy_scan@GLIBC_2.2.5	extern
__libc_start_main	extern
strcpy	extern
__strcpy_scan	extern
__strcpy_scan@GLIBC_2.2.5	extern
__strcpy_scan@GLIBC_2.2.5	extern

IDA - Graph View



IDA - Pseudo Code

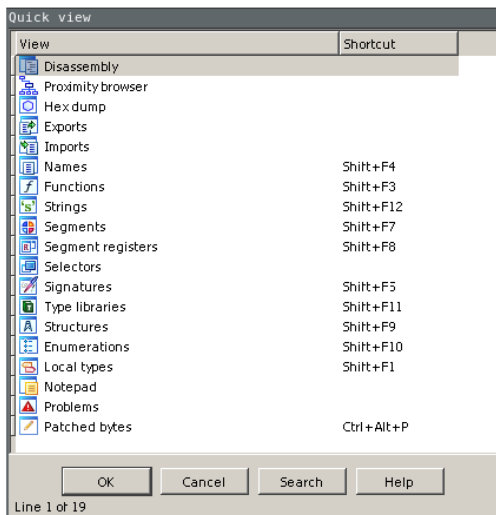
The screenshot shows the IDA Pro interface with the following components:

- Menu Bar:** File, Edit, Jump, Search, View, Debugger, Options, Windows, Help.
- Toolbar:** Standard IDE navigation and execution tools.
- Functions window:** A list of functions with their segments. The 'main' function is highlighted in the 'text' segment.
- Code View:** The pseudo-code for the 'main' function is displayed in the IDA View-A window.

```

1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     const char *v3; // ST1030_001
4
5     v3 = (const char *)calloc(1uLL, 0x80uLL);
6     puts("Welcome to our little password guessing game. Give it a try:");
7     __isoc99_scanf("%s", v3);
8     if ( (unsigned int)check_password(v3) )
9         puts("You won! :)");
10    else
11        puts("You lost :)");
12    return 0;
13 }
  
```

IDA - Possible Views



Demo Programm

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[]){
    char* input;
    input = (char*) calloc(1,128);
    printf("Welcome_to_our_little_password_guessing_game._Give_it_a_try:\n");

    scanf("%s", input);

    if(check_password(input)){
        printf("You_won!_:\n");
    }else{
        printf("You_lost_:(\n");
    }
    return 0;
}
```

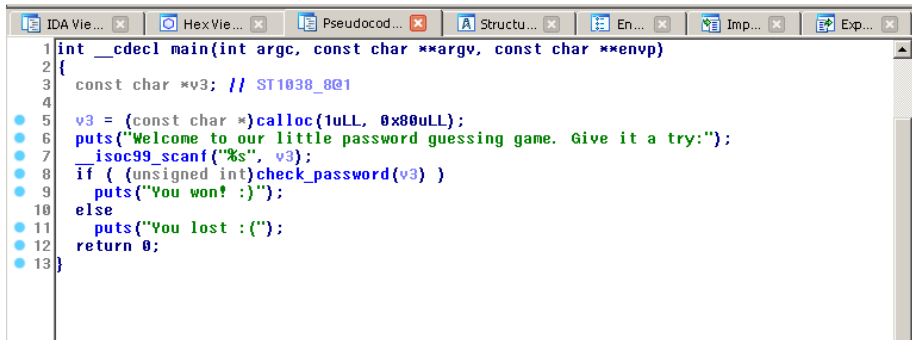

gdb - Beispiel



GDB

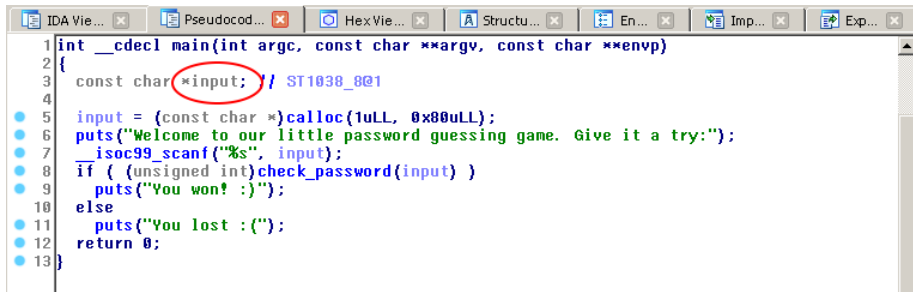
The GNU Project
Debugger

IDA - Beispiel



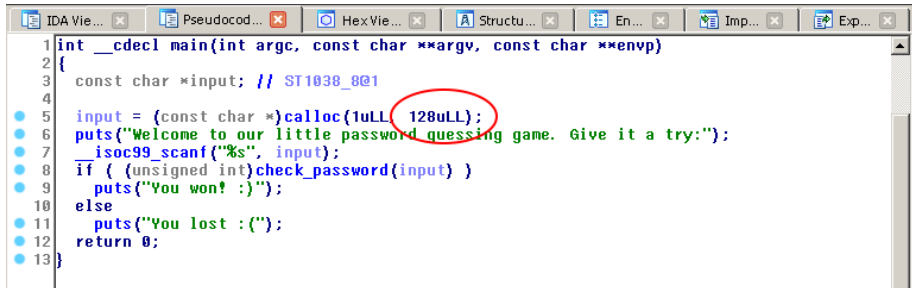
```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     const char *v3; // ST1038_8@1
4
5     v3 = (const char *)calloc(1uLL, 0x80uLL);
6     puts("Welcome to our little password guessing game. Give it a try:");
7     __isoc99_scanf("%s", v3);
8     if ( (unsigned int)check_password(v3) )
9         puts("You won! :)");
10    else
11        puts("You lost :(");
12    return 0;
13}
```

IDA - Beispiel



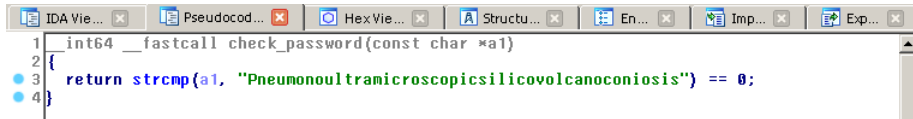
```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3   const char *input; // ST1038_8@1
4
5   input = (const char *)calloc(1uLL, 0x80uLL);
6   puts("Welcome to our little password guessing game. Give it a try:");
7   __isoc99_scanf("%s", input);
8   if ( (unsigned int)check_password(input) )
9     puts("You won! :)");
10  else
11    puts("You lost :(");
12  return 0;
13 }
```

IDA - Beispiel



```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     const char *input; // ST1038_8@1
4
5     input = (const char *)calloc(1uLL, 128uLL);
6     puts("Welcome to our little password guessing game. Give it a try:");
7     _isoc99_scanf("%s", input);
8     if ( (unsigned int)check_password(input) )
9         puts("You won! :)");
10    else
11        puts("You lost :(");
12    return 0;
13 }
```

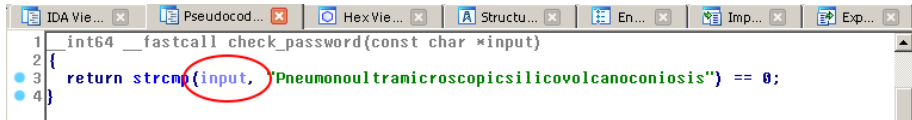
IDA - Beispiel



The screenshot shows the IDA Pro interface with several tabs open: IDA Vie..., Pseudocod..., Hex Vie..., Structu..., En..., Imp..., and Exp... The main window displays the following C code:

```
1 int64 __fastcall check_password(const char *a1)
2 {
3     return strcmp(a1, "Pneumonoultramicroscopicsilicovolcanoconiosis") == 0;
4 }
```

IDA - Beispiel



```
1  __int64 __fastcall check_password(const char *input)
2  {
3  return strcmp(input, "Pneumonoultramicroscopicsilicovolcanoconiosis") == 0;
4  }
```

Methoden gegen Reverse Engineering

Methoden gegen Reverse Engineering

- Stripped Binaries
 - Binärdateien ohne Debug Symbole
 - Unter anderem Funktionsnamen gehen verloren

Methoden gegen Reverse Engineering

- Stripped Binaries
 - Binärdateien ohne Debug Symbole
 - Unter anderem Funktionsnamen gehen verloren
- Code Obfuscation
 - Code möglichst kompliziert und unverständlich gestalten
- Aladdin HASP HL Pro

Methoden gegen Reverse Engineering

- Stripped Binaries
 - Binärdateien ohne Debug Symbole
 - Unter anderem Funktionsnamen gehen verloren
- Code Obfuscation
 - Code möglichst kompliziert und unverständlich gestalten
- Aladdin HASP HL Pro
 - USB Dongle um Software zu verifizieren
- Envelope Protection

Methoden gegen Reverse Engineering

- Stripped Binaries
 - Binärdateien ohne Debug Symbole
 - Unter anderem Funktionsnamen gehen verloren
- Code Obfuscation
 - Code möglichst kompliziert und unverständlich gestalten
- Aladdin HASP HL Pro
 - USB Dongle um Software zu verifizieren
- Envelope Protection
 - Verschlüsselter Code

Methoden gegen Reverse Engineering

- Stripped Binaries
 - Binärdateien ohne Debug Symbole
 - Unter anderem Funktionsnamen gehen verloren
- Code Obfuscation
 - Code möglichst kompliziert und unverständlich gestalten
- Aladdin HASP HL Pro
 - USB Dongle um Software zu verifizieren
- Envelope Protection
 - Verschlüsselter Code
- Cloud-Based-Software

Tools gegen Reverse Engineering I

Themida

- Software Protector umschließt die eigentliche, verschlüsselte Anwendung
- Bevor der Protector die Anwendung entschlüsselt stellt er sicher, dass keinerlei "Cracking Software" aktiv ist
- Themida behauptet alle gängigen Tools erfolgreich erkennen zu können
- Kann Probleme mit Antiviren Software hervorrufen

Tools gegen Reverse Engineering II

Code Virtualizer

- Programm beinhaltet eine virtuelle CPU auf der es ausgeführt wird
- Eigentlicher Code besteht aus Maschinen-Code der nur von der virtuellen CPU verstanden wird
- Jede virtuelle CPU kennt unterschiedliche Befehle, das selbe Programm sieht bei jedem Kunden anders aus

Tool Sammlung

Tool Sammlung

- edb
 - GUI für gdb

Tool Sammlung

- edb
 - GUI für gdb
- Radare2
 - Commandline Reverse Engineering Framework
 - Sowohl statische als auch dynamische Analyse
 - Komplexe Bedienung
 - Schöne Tree-View und gute Gesamtübersicht

Tool Sammlung

- edb
 - GUI für gdb
- Radare2
 - Commandline Reverse Engineering Framework
 - Sowohl statische als auch dynamische Analyse
 - Komplexe Bedienung
 - Schöne Tree-View und gute Gesamtübersicht
- Hopper
 - IDA's kleiner Bruder
 - Kann Pseudo Code aus Binärdateien erstellen
 - Bei weitem nicht so gut wie IDA
 - 30 Minuten gratis Demo verfügbar

Vielen Dank für eure Aufmerksamkeit!