re|view

reverse engineering

[binary file]

viewer

Open-Source Projekt

2011, Mario Schallner

Was ist re|view ?

```
xR 0805eb60 8B 0D 58 E7 10 08    mov    ecx, [shell_builtins]
   0805eb66 01 F1                add    ecx, esi
   0805eb68 83 C6 18             add    esi, 0x18
   0805eb6b 8B 01                mov    eax, [ecx]
   0805eb6d 85 C0                test   eax, eax
   0805eb6f 74 6F                jz     0x0805EBE0
```

Open-Source Projekt

Low-Level Binary-Analyseprogramm

- Interaktiver Disassembler
- Integrierte abstrakte Programm-Code Suche
- Programmierbar (→ acss, → rv|script)
- Flexibles User Interface

Einsatzgebiete

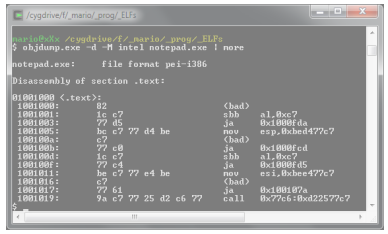
- Automatisierte / manuelle Low-Level Code-Analyse, Binary Auditing, Malware-Analyse (Viren, Trojaner, Rootkits, Exploits, ...), Ausbildung

Programmiert in C++ / Qt: ~ 17.000 Zeilen Quelltext

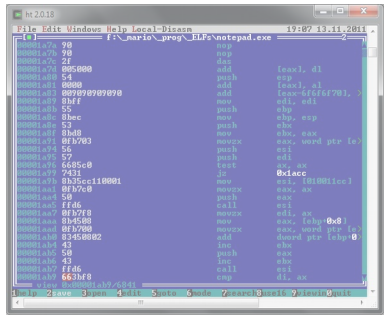
```
65      int disassemble_flow_recursive(
66              rv_addr_t va,
67              QMap<rv_addr_t, char> *map,
68              int &reclevel, rv_addr_t va_from);
69
70      QMap<rv_addr_t, char> *disassemble_flow_recursive_from(
71              rv_addr_t va, QMap<rv_addr_t, char> *M);
72
73      int disassemble_single(rv_addr_t va, x86_insn_t *insn);
74
75      // -- post disassembly functions --
76      QHash<rv_addr_t, rv_addr_t> *process_disassembly_caves(
77              QMap<rv_addr_t, char> *M);
78
79      QHash<rv_addr_t, rv_addr_t> *process_disassembly_hiddenInsns(
80              QMap<rv_addr_t, char> *M,
81              QHash<rv_addr_t, rv_addr_t> *allready_hidden = 0);
82
83      void format_insn( x86_insn_t *insn, char *buf, int len );
84      void format_insn4preview( x86_insn_t *insn, char *buf, int len );
85      void format_mnemonic( x86_insn_t *insn, char *buf, int len );
86      void format_operands( x86_insn_t *insn, char *buf, int len );
87      void format_src_operands( x86_insn_t *insn, char *buf, int len );
88      void get_dest_operands( x86_insn_t *insn, char *buf);
89      int check_dest_controlflow( x86_insn_t *insn );
90      int check_dest_memaccess( x86_insn_t *insn );
91      rv_addr_t rv_x86_get_address(x86_insn_t *i);
92
93      // acss use in disassembler directly
94      rv_addr_t search_deep(rv_acss_parser *p,
95                            rv_addr_t &va_from, rv_addr_t va_until);
96
97      char *disassemble_preview(rv_addr_t va_start,
98                            rv_addr_t va_end, char *rbuf);
99
100     // --------- for use on acss click -----------
101     QHash<rv_addr_t, rv_addr_t> *search_disassembly(
102             rv_acss_parser *p,
103             QMap<rv_addr_t, char> *va_insnsizes_map,
104             rv_addr_t va_from = 0, rv_addr_t va_until = 0,
105             int maxresults = 0);
106
107     QHash<rv_addr_t, rv_addr_t> *search_deep_all(
108             rv_acss_parser *p, int maxresults = 0);
109
110     QHash<rv_addr_t, rv_addr_t> *search_deep_range(
111             rv_acss_parser *p,
112             rv_addr_t va_from = 0, rv_addr_t va_until = 0,
113             int maxresults = 0
114     public slots:
115
116     signals:
117     void progress_changed_to(const int &);
118     void current_task(const QString &);
119
120     public:
121     QMultiMap<rv_addr_t, rv_addr_t> *va_xrefs_mmap;
122     QString func_reg_acss_filter;
123
124     rv_address_descriptor *disasm_ad;
125     rv_file               *disasm_file;
126     // TODO: remove rv_file parameter from function calls !
127
128 };
129
130 #endif // DISASMBUFFER_H
```

Motivation zu re|view

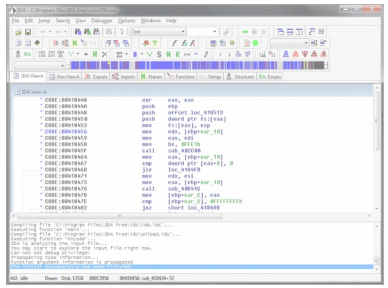
Das perfekte Tool hört nicht da auf, wo es endet



In der Malwareanalyse hat man sehr viel mit kompaktem, kreativen (speziell gestalteten) Low-Level Code zu tun. Disassembling ist eine Grundvoraussetzung zur Analyse. Gute Tools sind sehr teuer und machen nie genau das was man möchte, bzw. alles das, was man braucht.

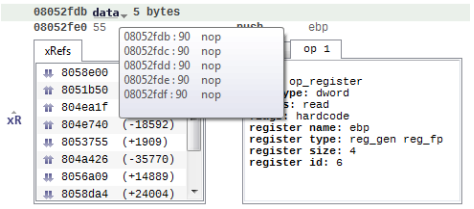


Eigene Tools für jeden spezielleren Anwendungsfall zu schreiben ist aufgrund der Komplexität des Themas keine Option.



re|view soll Abhilfe schaffen. Freie Software. Von jedem erweiterbar. Flexibel aufgebaut, bietet es eine Vielzahl an Möglichkeiten komfortabel Low-Level Code zu analysieren – bis hin dazu, in wenigen Zeilen Script seinen eigenen „special purpose“ Disassembler zu bauen (!).

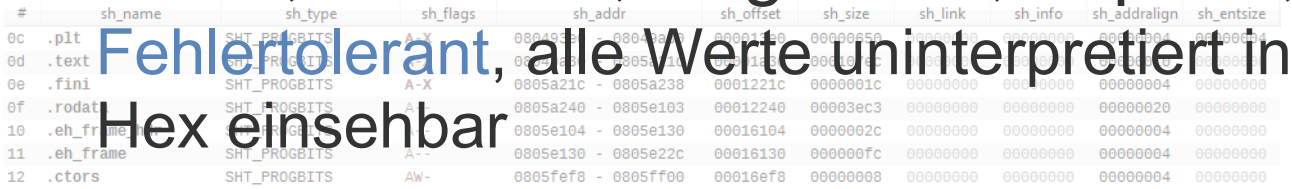
Funktionsumfang – Teil I



Läuft auf MS Windows, Linux, Mac OSX

Analyse von ELF32 (Linux), PE32 (MS Windows) Executables, RAW Dateien

Robuste Analyse der Datei-Struktur: Standard Headers, Sections, Segments, Imports, ...

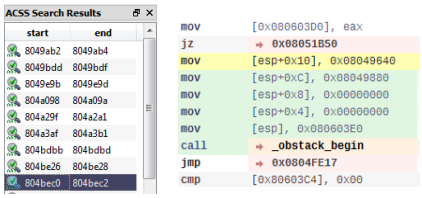


Fehlertolerant, alle Werte uninterpretiert in Hex einsehbar

Interaktiver Disassembler

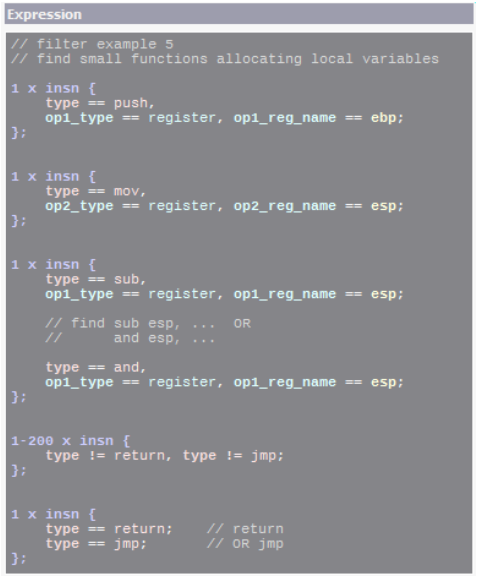
Control Flow Analyse, Crossreferences, String Data References, dynamische Detail-Views, Programmierbar ... ...

Funktionsumfang – Teil II



Abstract Code Sequence Search:

Eigens entwickelte Filtersprache zur abstrakten Suche nach Code-Sequenzen und Algorithmen, extrem schnell durch binäre on-the-fly Kompilierung



rv|script: Objektorientierte Skriptsprache

ECMA basierend („JavaScript“)

Zugriff auf Daten und Funktionen des Disassemblers, Instructions Objects, ... (disassemble_insn(), xrefs_at(), ...)

Besonderheiten von re|view – Teil I

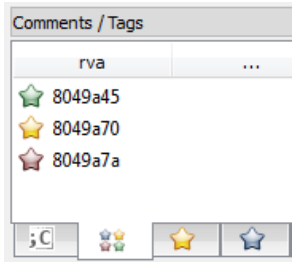
Freie Software  

Speziell entwickelt für Anforderungen der Malware-Analyse:

- Instruction-Metadaten-Suche (acss) im Disassembly, oder über gesamtes Binary:
  - → auch in Datenblöcken
- acss kann zur abstrakten Suche von Algorithmen eingesetzt werden, wo Signatur-Scanning versagt
- Umgang mit „hostile binaries“ (korrupte Headers, anti-disassembling, ...)

Besonderheiten von re|view – Teil II

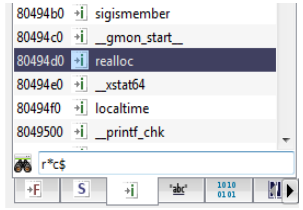
Schwerpunkt komfortables und flexibles Us
Interface:



- Dynamische Detail-Views, Dock Windows
- Navigation mittels Tags, Comments, Names, ...
- Integrierte Programmier-Möglichkeiten (Programmierung und Verarbeitung der Ergebnisse im GUI möglich)

```
var addr = lookup_function_rva("entry_point");

var insn;

for(var i=0; i<MAX; i++) {
    insn = disassemble_insn(addr);
    if(insn.dest_rva == 0x80496b0) {
        msg(insn.str);
        for(var j=0; j<insn.size;j++)
        patch_byte(addr + j, 0x90);
    }
    addr += insn.size;
    if(!insn.size) addr++;
}
```

- Syntax-Highlighting, Regular Expression Search in allen Suchfenstern

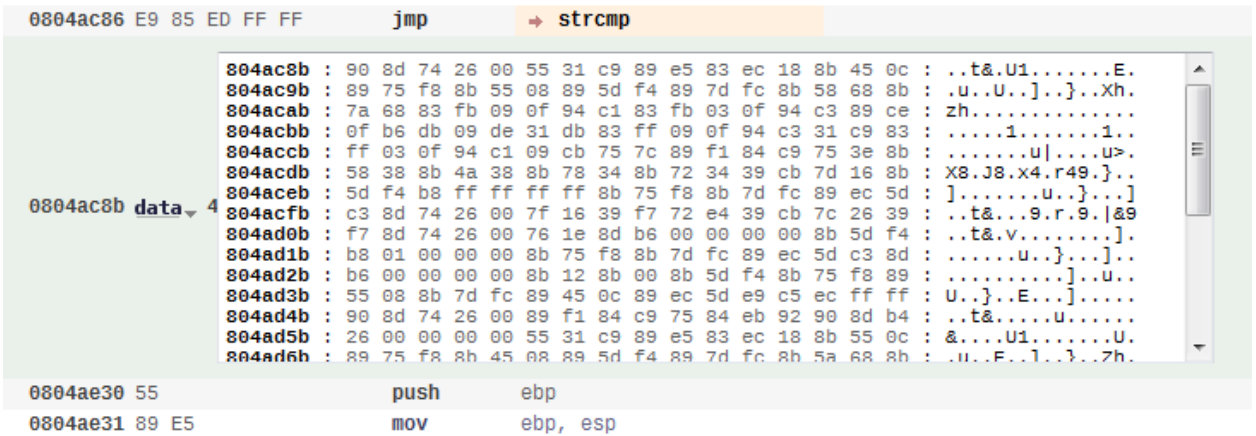


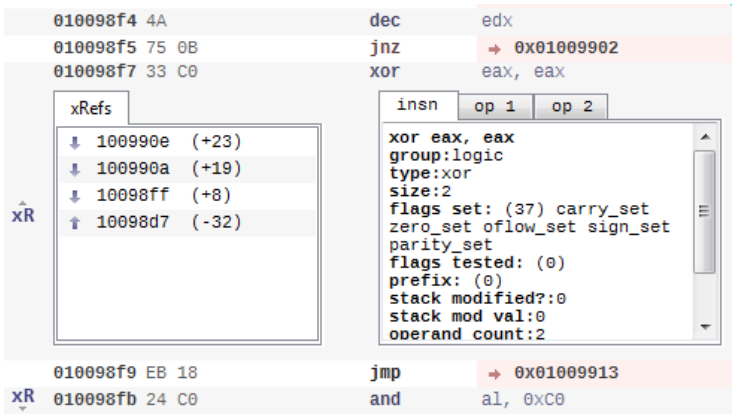
- „Command-Line“: Direkte Befehls-eingabe im GUI

Besonderheiten r|v – Disassembler

Spezielle Darstellungsform für die komfortable Analyse
- Strings im Klartext (`mov [esp+0x4], "../bash/shell.c"`),
- Daten bei indirekter Addressierung im Disassembly dargestellt (`mov eax, [0x080602A0] ; [0x080602A0] = 1`), ...

Dynamische Darstellung von Metadaten,
- Crossreferences, Datenblöcken ...
- on demand, direkt im Disassembly:





Besonderheiten r|v – Disassembler

- Alle Address-Parameter „clickable“, auch bei non-controlflow instructions (zB.: `push → 0x0805A140`)
- Darstellung von File-Offset und Virtual Address
- Darstellung von in Instructions codierten ASCII Strings direkt im Disassembly (zB.:

```
0805f3c5 C7 00 2F 62 69 6E    mov       [eax], 0x6E69622F
0805f3c7                                                  /bin
```

)

  - zB. zur Erkennung von Shellcodes

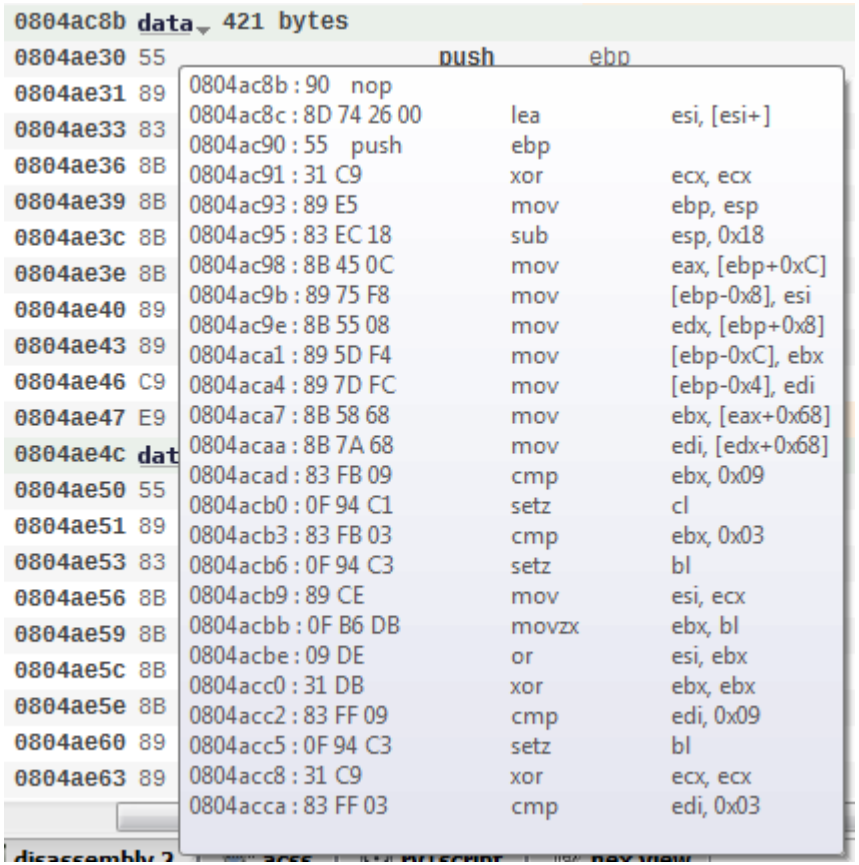
- Control Flow Analyse erkennt u. markiert disaligned Instructions:

```
         08048060 E8 01 00 00 00      call      → 0x08048066
         08048065 E9 58 90 05 0B      jmp       0x130A10C2
   xR    08048066 58                  pop       eax
         08048067 90                  nop
         08048068 05 0B 00 00 00      add       eax, 0x0000000B
         0804806d 50                  push      eax
         0804806e C3                  ret
         0804806f data 16 bytes
```

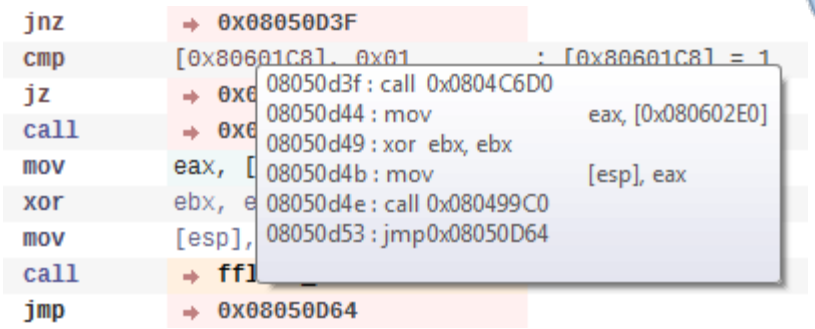
Besonderheiten r|v – Disassembler

Tooltip Disassemblies für Datenblöcke und Sprungziele:

Datenblock

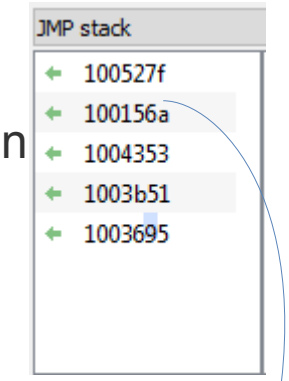


Sprungziel

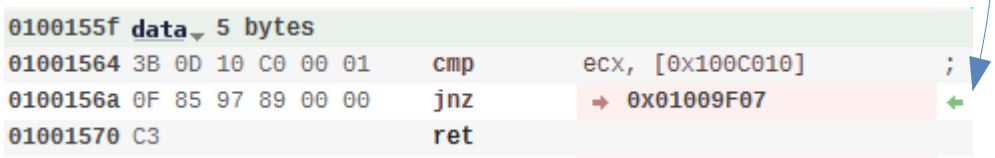


- **JMP Stack**

speichert Rücksprungsaddressen während der Navigation:



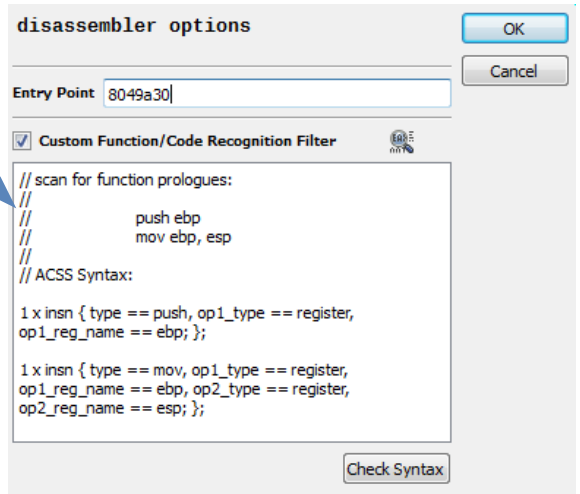
Markierung der Adressen im Disassembly:



Besonderheiten r|v – Disassembler

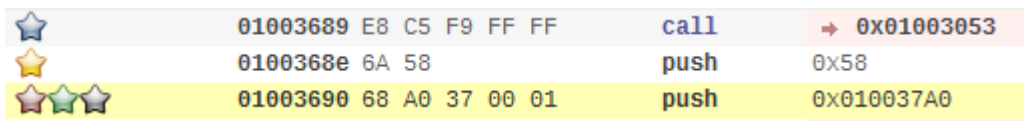
Disassembler benutzt „acss“ zur Analyse des Binaries,

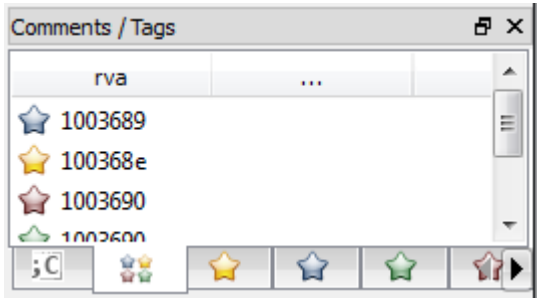
parametrisierbarer (!) Function-Recognition Filter zum Aufspüren von nicht-referenzierten Code-Blöcken



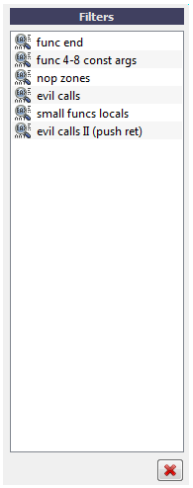
Steuerbar über Commandline – Befehle

Direktes Taggen von Positionen im Disassembly





r|v Metadatensuche – ACSS

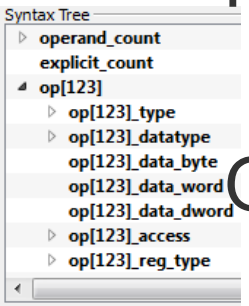


Filtersprache zur abstrakten Formulierung von Code-Sequenzen und Algorithmen:

```
// filter example 4
// evil calls I
// call register

1 x insn {
    op1_type == register,
    op1_access == (&execute);
};
```

```
// filter example 2

4-8 x insn {
    group == move,
    op2_type == immediate,
    op2_datatype == dword;
};

1 x insn { type == call; };
```

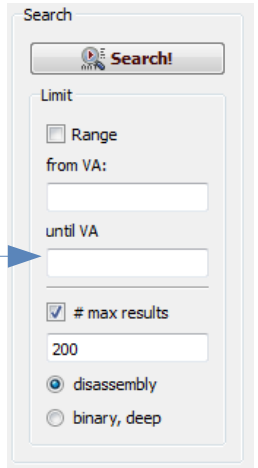
```
1 x insn {
    type == sub,
    op1_type == register, op1_reg_name == esp;

    // find sub esp, ...  OR
    //      and esp, ...

    type == and,
    op1_type == register, op1_reg_name == esp;
};


1-200 x insn {
    type != return, type != jmp;
};
```



Metadaten inkludieren uA.: instruction group, type, flags-status, operands, registers, data, access type, ...

Operatoren: ==, !=, implizites AND und OR (über Struktur)

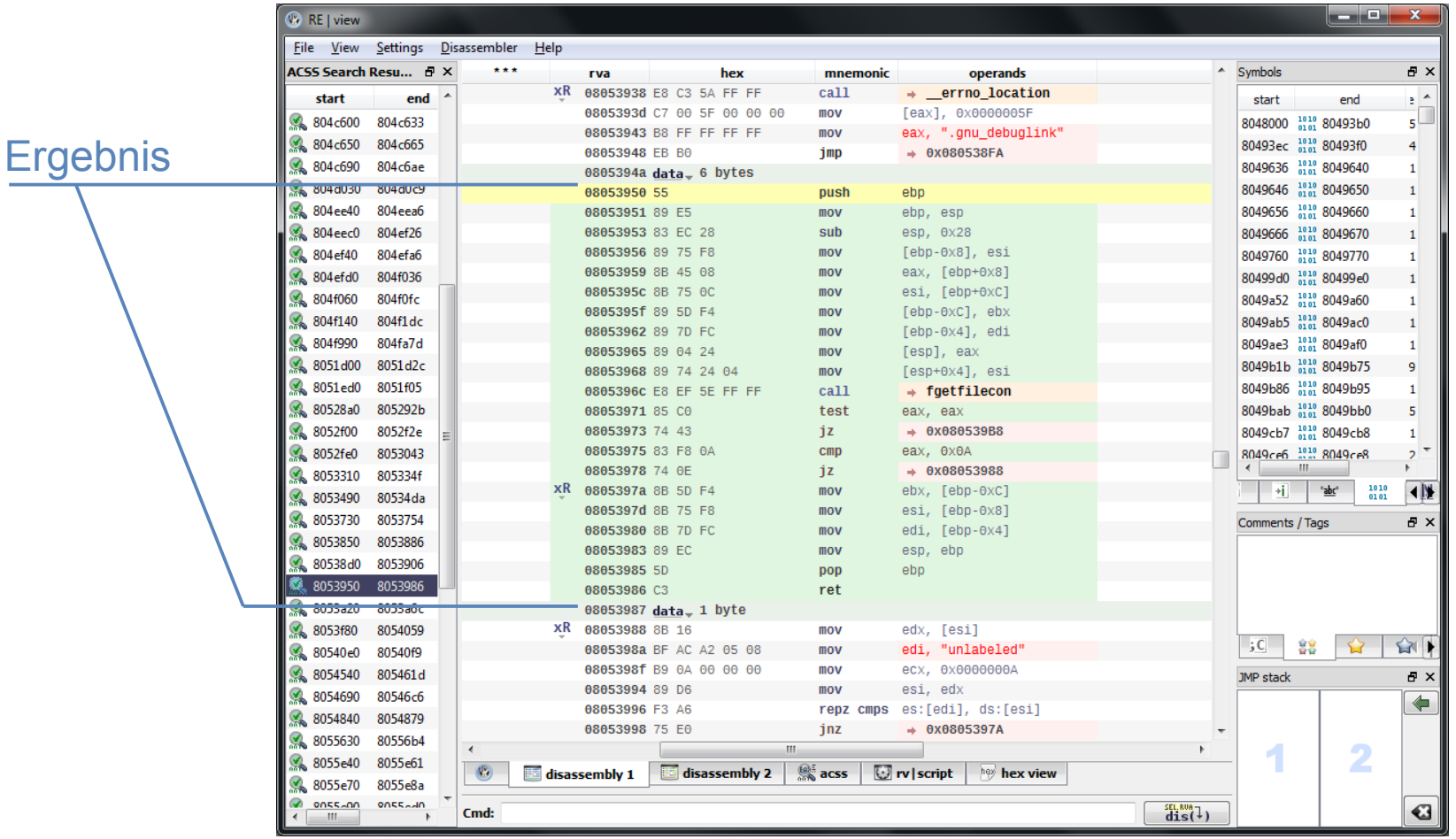
Suche im Disassembly, oder über gesamtes Binary möglich, Bereichsangaben, etc ...



Filter organisiert in Libraries, werden in binäre Form kompiliert extrem schnelle Suche

r|v Metadatensuche – ACSS

Suchergebnisse über GUI zugreifbar und hinterlegt:



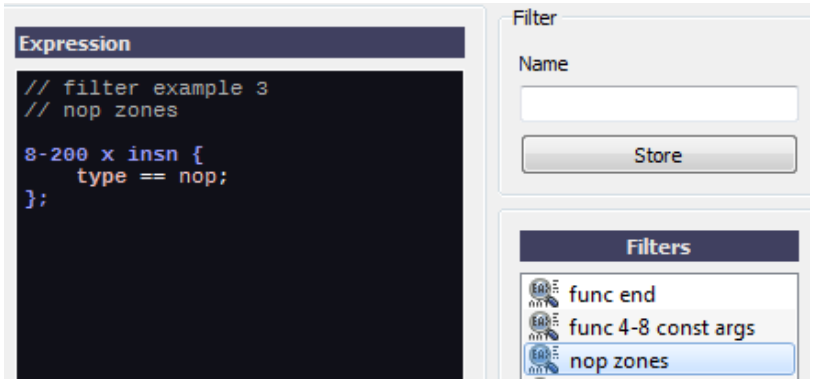
r|v Metadatensuche – ACSS

Möglichkeit acss auch ohne Disassembly (ohne vorherige Control-Flow Analyse) einzusetzen:

Suchergebnisse werden rot hinterlegt, bei Doppelklick wird vom Ergebnis weg ein Control-Flow Analyse gestartet, und Ergebnis grün hinterlegt

D.h. Es wurden nur die Instructions verfolgt, die von diesem Code-Teil aus erreicht werden. Alles andere bleibt ausgeblendet (!)

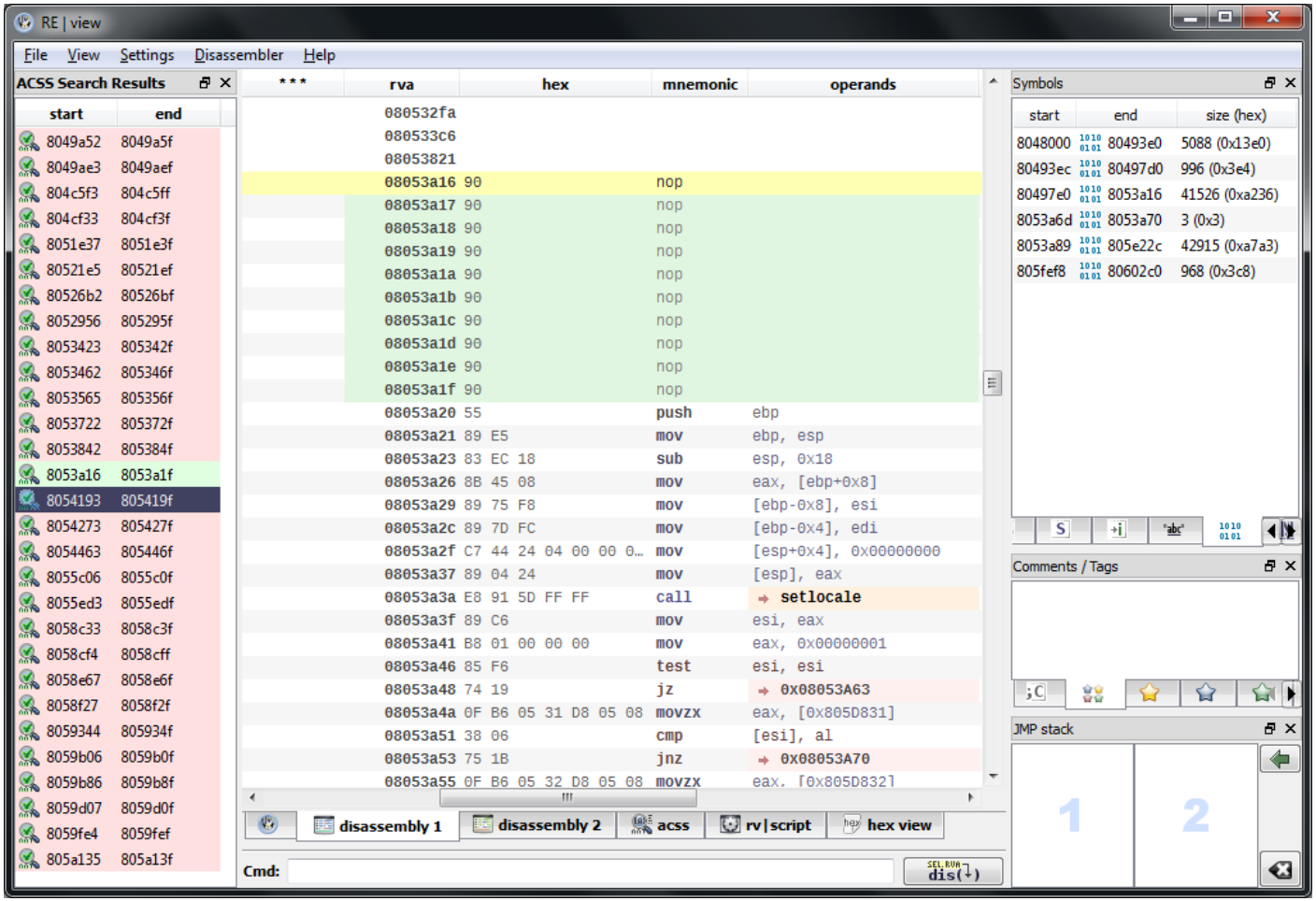
zB:



auf leeres Disassembly angewendet:

r|v Metadatensuche – ACSS

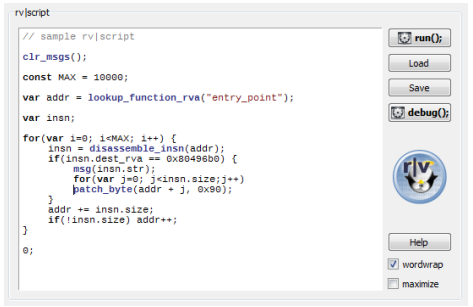
Suchergebnis 0x8053a16:

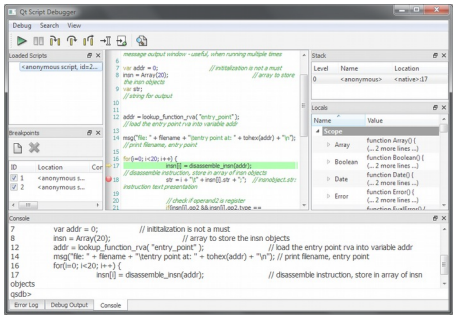


Erweiterung mit r|v script

Objektorientierte Programmiersprache
nach dem ECMA Standard („JavaScript“)

inklusive Debugger





Bietet Zugriff auf die Disassembling-Engine

Stellt Objekte des Typs „Instruction“ zur
Verfügung, welche alle Metadaten enthalten

Stellt Funktionen der Disassembling-Engine zur Verfügung:
zB: disassemble_insn(), lookup_function_rva(), ...

Stellt Listen-Objekte für Crossreferences zur Verfügung
zu füllen per xrefs_at();

Syntax Highlighting, Ausgabefenster für Meldungen, ...

r|v script

Beispielscript:

rv|script

```
// sample rv|script
// print the disassembly of the entrypoint, do demo checks on insn.group, insn.type, insn.op1.type,
// and store insns in an array (20 lines)

clr_msgs();           // clear the message output window - useful, when running multiple times

var addr = 0;         // inititalization is not a must
insn = Array(20);     // array to store the insn objects
var str;              // string for output


addr = lookup_function_rva( "entry_point" );       // load the entry point rva into variable addr

msg("file: " + filename + "\tentry point at: " + tohex(addr) + "\n"); // print filename, entry point

for(i=0; i<20; i++) {
    insn[i] = disassemble_insn(addr);      // disassemble instruction, store in array of insn objects
    str = i + "\t" + insn[i].str + ";";    // insnobject.str: instruction text presentation

    // check if operand2 is register
    if(insn[i].op2 && insn[i].op2.type == OP_REGISTER)  str += "\t(insn.op2.reg.name: '" + insn[i].op2.reg.name + "')";

    // check if operand1 is immediate value
    if(insn[i].op1 && insn[i].op1.type == OP_IMMEDIATE) str += "\t(insn.op1.data (immediate): " + tohex(insn[i].op1.data) + ")";

    if(insn[i].group == INSN_LOGIC) str += "\tinsn.group: LOGIC !";
    if(insn[i].type == INSN_CALL) str += "\tinsn.type: CALL !";

    msg(str);
    addr += insn[i].size;         // set addr to rva of next instruction
}

msg("\n\n == insn[5] ==:\n" + insn[5].str);

0;
```

run();
Load
Save
debug();
Help
wordwrap
maximize

r|v script

Ausführung des Beispielscripts:

messages

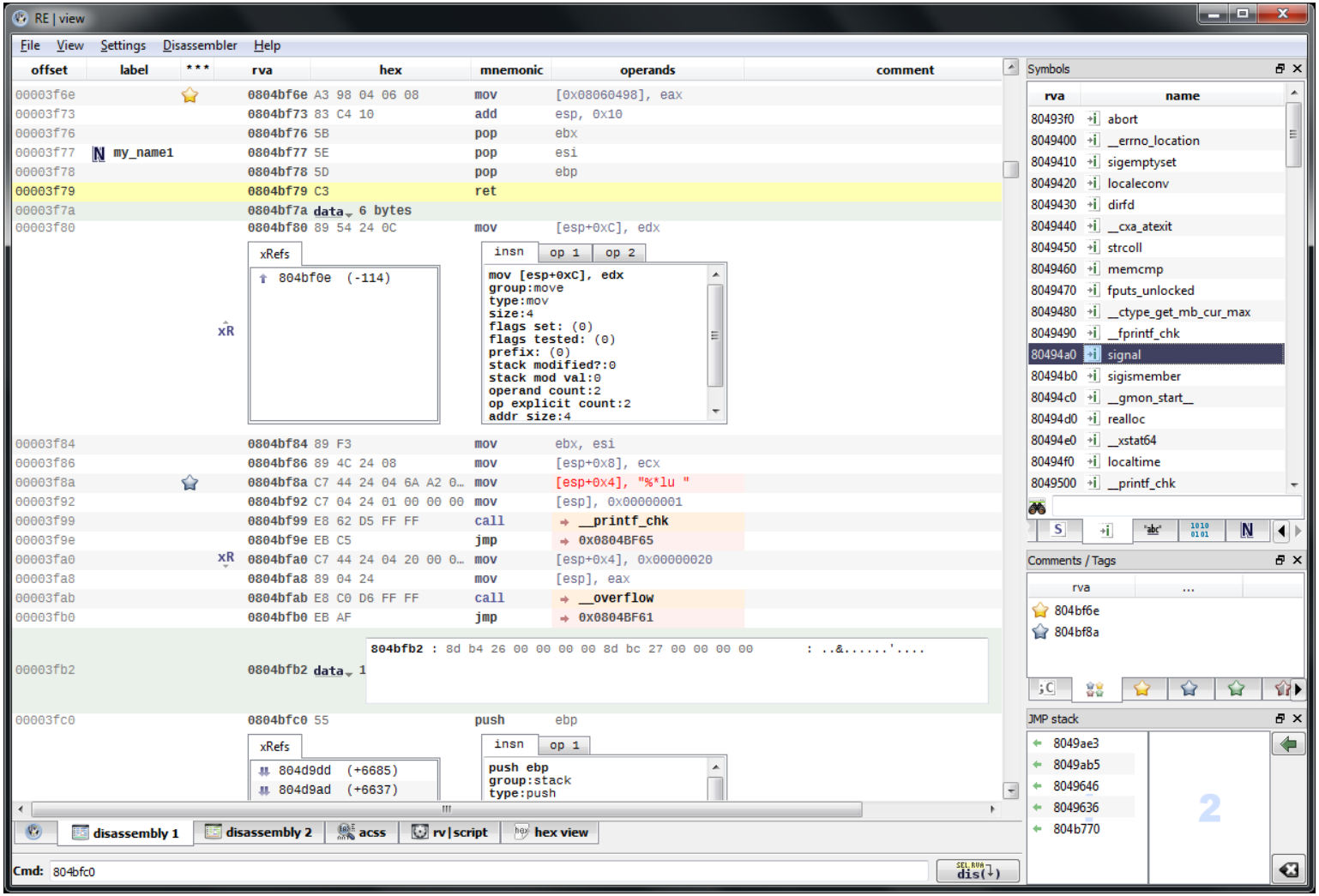
```
file: F:/_mario/_prog/_ELFs/ls    entry point at: 8049A30

0   08049a30 : xor   ebp, ebp;    (insn.op2.reg.name: 'ebp')    insn.group: LOGIC !
1   08049a32 : pop   esi;
2   08049a33 : mov   ecx, esp;    (insn.op2.reg.name: 'esp')
3   08049a35 : and   esp, 0xF0;   insn.group: LOGIC !
4   08049a38 : push  eax;
5   08049a39 : push  esp;
6   08049a3a : push  edx;
7   08049a3b : push  0x0805A1A0;  (insn.op1.data (immediate): 805A1A0)
8   08049a40 : push  0x0805A140;  (insn.op1.data (immediate): 805A140)
9   08049a45 : push  ecx;
10  08049a46 : push  esi;
11  08049a47 : push  0x0804F990;  (insn.op1.data (immediate): 804F990)
12  08049a4c : call  0x08049580;  insn.type: CALL !
13  08049a51 : hlt   ;
14  08049a52 : nop   ;
15  08049a53 : nop   ;
16  08049a54 : nop   ;
17  08049a55 : nop   ;
18  08049a56 : nop   ;
19  08049a57 : nop   ;


 == insn[5] ==:
08049a39 : push  esp
--- script returned: '0'
--- script terminated ---
```

Clear

Save

maximize

re|view



re|view