Finding library subroutines in stripped statically-linked binaries

findmagic

Katharina Bogad
Technische Universität München
Computer Science Department

SS 2015
January 18, 2017

obligatory tl;dr me slide

- Computer Science student
- Member of the H4x0rPsch0rr CTF-Team and CTF-Player for fun (and sometimes profit)
- Interested in reverse engineering for long time
- Hates QR-Codes

Preliminary audience questions

Who of you has...

- basic knowledge of graph theory?

Preliminary audience questions

Who of you has...

- basic knowledge of graph theory?
- reverse engineered a statically linked binary at least once?

Problem description

Why?

- Traditional pattern-matching: exact library needed for decent results
- Works reasonably well in homogenous environments like MSVCRT
- Open source libraries?
- Embedded devices?

Problem description

Why?

So, what are we doing if we cannot have symbols?

- Looking at the arguments?
- Looking at suspicious constants?
  Think of `0x8080808080` for `strlen(3)`

Let's automate this!

However, there are caveats:

- Finding arguments is not a trivial task.
- What makes a constant suspicious?

Problem description

Why?

However, there are caveats:

- Finding arguments is not a trivial task.
- What makes a constant suspicious?

But automating gives new perspectives:
Comparing callgraphs!

- Program is a set of attributed graphs $G = (N, B)$
- Nodes N are functions
- Branches B are calls between functions

We need:

- A string definition

$$
\begin{aligned}
&(\forall (i,c) \in str : c \geq \texttt{0x20} \wedge c \leq \texttt{0xDF} \\
&\quad \vee c = \texttt{0x0A} \vee c = \texttt{0x0D} \vee c = \texttt{0x09} \vee c = \texttt{0x00}) \\
&\quad \wedge |str| > 1 \\
&\quad \wedge (\forall (i,c) \in str \,|\, i = \max(i, str) : c = \texttt{0x00}) \\
&\quad \wedge (\forall (i,c) \in str \,|\, i \neq \max(i, str) : c \neq \texttt{0x00})
\end{aligned} \tag{1}
$$

We need:

- A string definition

$$
\begin{aligned}
&(\forall (i,c) \in str : c \geq \mathtt{0x20} \land c \leq \mathtt{0xDF} \\
&\quad \lor c = \mathtt{0x0A} \lor c = \mathtt{0x0D} \lor c = \mathtt{0x09} \lor c = \mathtt{0x00}) \\
&\quad \land |str| > 1 \\
&\quad \land (\forall (i,c) \in str \,|\, i = \max(i, str) : c = \mathtt{0x00}) \\
&\quad \land (\forall (i,c) \in str \,|\, i \neq \max(i, str) : c \neq \mathtt{0x00})
\end{aligned} \tag{1}
$$

Printable characters from extended ASCII ...

We need:

- A string definition

$$
\begin{aligned}
&(\forall (i,c) \in str : c \geq \texttt{0x20} \wedge c \leq \texttt{0xDF} \\
&\quad \vee c = \texttt{0x0A} \vee c = \texttt{0x0D} \vee c = \texttt{0x09} \vee c = \texttt{0x00}) \\
&\quad \wedge |str| > 1 \\
&\quad \wedge (\forall (i,c) \in str \,|\, i = \max(i, str) : c = \texttt{0x00}) \\
&\quad \wedge (\forall (i,c) \in str \,|\, i \neq \max(i, str) : c \neq \texttt{0x00})
\end{aligned} \tag{1}
$$

... and `\n`, `\r`, `\t` and `0x00` ...

We need:

- A string definition

$$
\begin{aligned}
&\big(\forall (i,c) \in str : c \geq \texttt{0x20} \wedge c \leq \texttt{0xDF} \\
&\quad \vee c = \texttt{0x0A} \vee c = \texttt{0x0D} \vee c = \texttt{0x09} \vee c = \texttt{0x00}\big) \\
&\quad \wedge |str| > 1 \\
&\quad \wedge \big(\forall (i,c) \in str \,|\, i = \max(i, str) : c = \texttt{0x00}\big) \\
&\quad \wedge \big(\forall (i,c) \in str \,|\, i \neq \max(i, str) : c \neq \texttt{0x00}\big)
\end{aligned} \tag{1}
$$

... with a minimum length of 2 ...

We need:

- A string definition

$$
\begin{aligned}
&\big(\forall (i,c) \in str : c \geq \texttt{0x20} \wedge c \leq \texttt{0xDF} \\
&\quad \vee c = \texttt{0x0A} \vee c = \texttt{0x0D} \vee c = \texttt{0x09} \vee c = \texttt{0x00}\big) \\
&\quad \wedge |str| > 1 \\
&\quad \wedge \big(\forall (i,c) \in str \,|\, i = \max(i, str) : c = \texttt{0x00}\big) \\
&\quad \wedge \big(\forall (i,c) \in str \,|\, i \neq \max(i, str) : c \neq \texttt{0x00}\big)
\end{aligned} \tag{1}
$$

... where the last character is `0x00` and no other character is `0x00`.

We need:

- A node definition
  $N = (n, s, C, S, I)$
  - n: Function name
  - s: Function address
  - C: Multiset of constant values
  - S: Multiset of cross-referenced strings
  - I: Ordered multiset of the machine instructions

Objective: Generate a bijective mapping $M = N_1 \to N_2$

- N_1: known library function
- N_2: function inside the target library

1. Acquire target library with debug symbols

1. Acquire target library with debug symbols
2. Build the graphs for it

Algorithm Design

Get crackin'

1. Acquire target library with debug symbols
2. Build the graphs for it
3. Build graphs for the binary we analyse

1. Acquire target library with debug symbols
2. Build the graphs for it
3. Build graphs for the binary we analyse
4. Match them

Do we need exactly the same binary used for linking?

- Short answer: no.

Do we need exactly the same binary used for linking?

- Short answer: no.
- Long answer: it depends.

- A reasonably close version is enough
- Watch out for compiler flags
- Also problematic: assert()

Caution: real world example

```
assert((unsigned long) (old_size) < (unsigned long) (nb + MINSIZE));
```

with relocation:

```
(unsigned long) (old_size) < (unsigned
     long) (
  nb + (unsigned long)(
    (((__builtin_offsetof (struct
         malloc_chunk, fd_nextsize)) +
        (
          (2 * (sizeof(size_t))) - 1
      ))
      & ~(
        (2 * (sizeof(size_t))) - 1
))))
```

No code, but debug strings vary!

without relocation:

```
(unsigned long) (old_size) < (unsigned
     long) (
  nb + (unsigned long)(
    (((__builtin_offsetof(struct
         malloc_chunk, fd_nextsize)) +
        ((2 * (sizeof(size_t)) <
             __alignof__ (long double) ?
           __alignof__ (long double) :
           2 * (sizeof(size_t))
         ) - 1))
      & ~(
        (2 * (sizeof(size_t)) <
             __alignof__ (long double) ?
          __alignof__ (long double) :
          2 * (sizeof(size_t))
        ) - 1
))))
```

1. Iterate over subroutines
2. Iterate over the instructions of these subroutines
3. If something interesting is found, add it to the corresponding list[1]

[1] See the paper for a marvellous formal definitions for this

- `call` instructions add a new branch to the functions callgraph
- Additionally for Intel x86_64 architecture:
- Only if it's a near call - opcode `0xE8`
- This ensures we're in the same section
- Other architectures may need different conditions!

- Look for something that loads a pointer (x86_64: `lea`, `mov`)
- Check if it's a string by our definition
- If so, add it to the Strings of the current function

- We don't want to add pointer arithmetic as constants
- Interesting constants are often bitmasks
- Thus, we limit ourselves to the immediates of `and`, `or`, `xor` and `mov`
- Optionally, we may exclude further by doing value checking on the constant

Isomorphism:

- Ancient greek: *isos* = equal and *morphe* = shape
- Mathematical way to compare the structure of objects

Choosing the right algorithm:

- Ullmann's algorithm
- Nauty (**n**o **aut**omporphism, **y**es?)
- VF2

Choosing the right algorithm:

- Ullmann's algorithm
- Nauty (**n**o **aut**omporphism, **y**es?)
- VF2

Choosing the right algorithm:

- Ullmann's algorithm
- Nauty (**n**o **aut**omporphism, **y**es?)
- VF2

Choosing the right algorithm:

- Ullmann's algorithm
- Nauty (**n**o **aut**omporphism, **y**es?)
- VF2

Choosing the right algorithm:

- Callgraphs cannot be considered randomly connected
- Some functions imply calls to other functions
- `malloc()` & `free()`, `accept()` & `close()`, ...
- VF2 is very fast in this situation
- Also, VF2 can check semantic equality of the nodes in the same step

VF2

- $G_1 = (N_1, B_1)$, $G_2 = (N_2, B_2)$
- Mapping $M \subset N_1 \times N_2$
- M must be a bijective function
- M must not alter the branch structure

- State Space Representation (SSR) s
- Essentially a set of tuples (n_1, n_2)
- $M(s)$ denotes a partial mapping
- Two subgraphs $G_1(s)$ and $G_2(s)$ can be derived, containing only the nodes in the matching and the branches connecting them
- Same for $M_1(s)$, $M_2(s)$, $B_1(s)$, $B_2(s)$

VF2

- Transition from state s to s': $s' = s \cup \{(n, m)\}$
- But: only a small set of these states are *consistent*
- We introduce *k-lookahead rules* to conclude wether a consistent state can be reached after k steps
- These rules will be called *feasibility rules*

Feasibility function: $F(s, n, m) = F_{syn}(s, n, m) \wedge F_{sem}(s, n, m)$

- $F_{syn} \rightarrow$ **syntactic** feasibility
- $F_{sem} \rightarrow$ **semantic** feasibility

- Initial state is empty, i.e. $M(s_0) = \varnothing$
- In each step, compute $P(S)$, the node pairs of candidates to be added
- $T_n^{in} \rightarrow$ nodes with branches ending into $G_n(s)$
- $T_n^{out} \rightarrow$ nodes with branches starting from $G_n(s)$
- $P(s) = \{(n, m) | n \in T_1^{out}, m \in T_2^{out}\}$ if no T_n^{out} is empty, T_n^{in} otherwise
- If $P(s)$ is still empty, backtrack until a state s is reached with $P(s)$ containing not examined node pairs

Check predecessors of current node:

$$R_{pred}(s,n,m) \iff$$
$$(\forall n' \in M_1(s) \cap \mathrm{Pred}(G_1,n)\, \exists\, m' \in \mathrm{Pred}(G_2,m) \mid (n',m') \in M(s)) \wedge$$
$$(\forall m' \in M_2(s) \cap \mathrm{Pred}(G_2,m)\, \exists\, n' \in \mathrm{Pred}(G_1,n) \mid (n',m') \in M(s))$$

Check successors of current node:

$$
\begin{aligned}
&R_{succ}(s,n,m) \iff \\
&(\forall n' \in M_1(s) \cap \mathrm{Succ}(G_1,n)\ \exists\, m' \in \mathrm{Succ}(G_2,m) \mid (n',m') \in M(s)) \wedge \\
&(\forall m' \in M_2(s) \cap \mathrm{Succ}(G_2,m)\ \exists\, n' \in \mathrm{Succ}(G_1,n) \mid (n',m') \in M(s))
\end{aligned}
$$

1-lookahead:

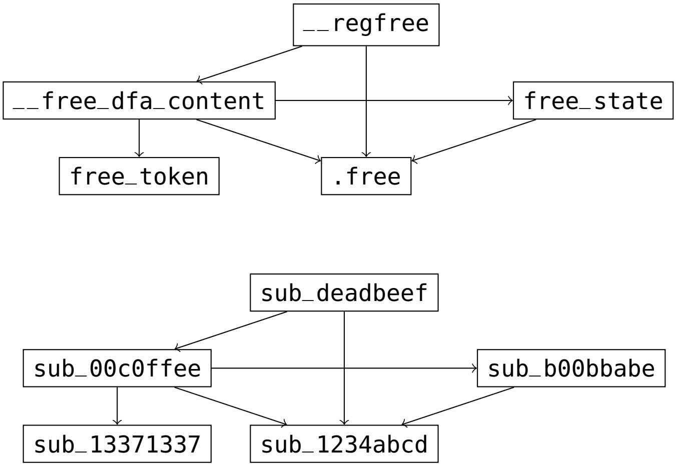
$$R_{in}(s,n,m) \iff$$
$$(\left|\mathrm{Succ}(G1,n) \cap T_1^{in}(s)\right| = \left|\mathrm{Succ}(G2,m) \cap T_2^{in}(s)\right|) \wedge$$
$$(\left|\mathrm{Pred}(G1,n) \cap T_1^{in}(s)\right| = \left|\mathrm{Pred}(G2,m) \cap T_2^{in}(s)\right|)$$

$$R_{out}(s,n,m) \iff$$
$$(\left|\mathrm{Succ}(G1,n) \cap T_1^{out}(s)\right| = \left|\mathrm{Succ}(G2,m) \cap T_2^{out}(s)\right|) \wedge$$
$$(\left|\mathrm{Pred}(G1,n) \cap T_1^{out}(s)\right| = \left|\mathrm{Pred}(G2,m) \cap T_2^{out}(s)\right|)$$

2-lookahead:

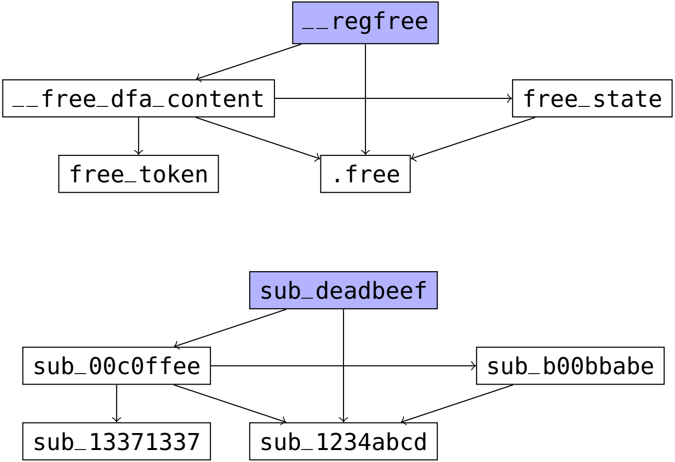
$$R_{new}(s,n,m) \iff$$
$$(\left|\widetilde{N}_1(s) \cap \mathrm{Pred}(G_1,n)\right| = \left|\widetilde{N}_2(s) \cap \mathrm{Pred}(G_2,m)\right|) \wedge$$
$$(\left|\widetilde{N}_1(s) \cap \mathrm{Succ}(G_1,n)\right| = \left|\widetilde{N}_2(s) \cap \mathrm{Succ}(G_2,m)\right|)$$

Example:



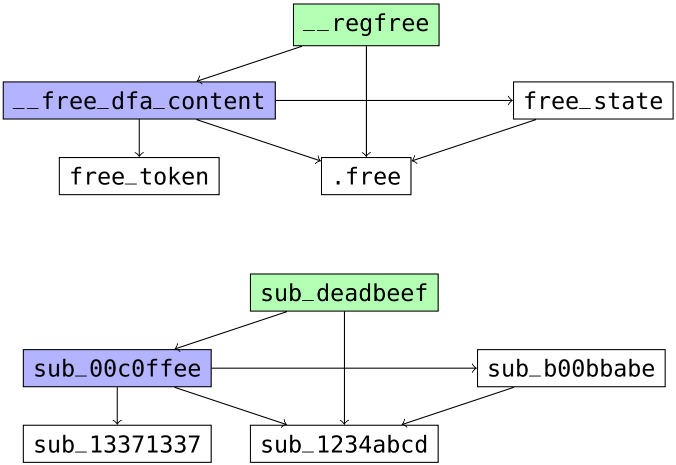
__regfree
__free_dfa_content
free_state
free_token
.free
sub_deadbeef
sub_00c0ffee
sub_b00bbabe
sub_13371337
sub_1234abcd

Example:

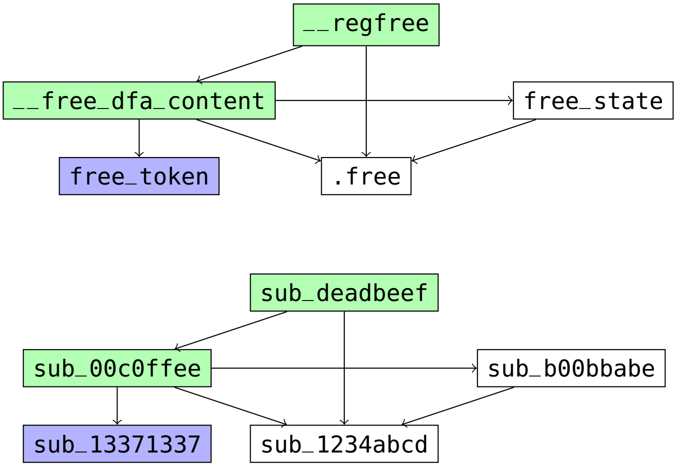


__regfree
__free_dfa_content
free_state
free_token
.free
sub_deadbeef
sub_00c0ffee
sub_b00bbabe
sub_13371337
sub_1234abcd

Example:

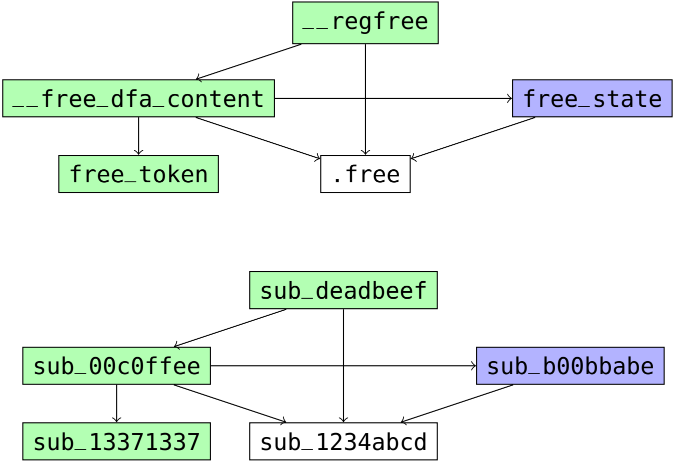

__regfree
__free_dfa_content
free_state
free_token
.free
sub_deadbeef
sub_00c0ffee
sub_b00bbabe
sub_13371337
sub_1234abcd

Example:

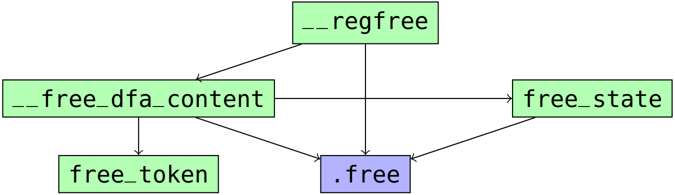


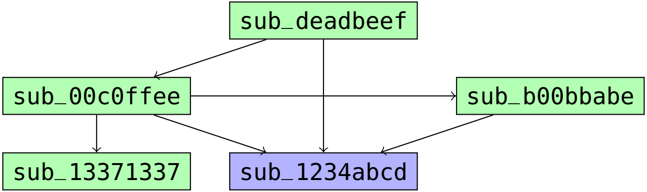
__regfree
__free_dfa_content
free_state
free_token
.free
sub_deadbeef
sub_00c0ffee
sub_b00bbabe
sub_13371337
sub_1234abcd

Example:


__regfree
__free_dfa_content
free_state
free_token
.free
sub_deadbeef
sub_00c0ffee
sub_b00bbabe
sub_13371337
sub_1234abcd

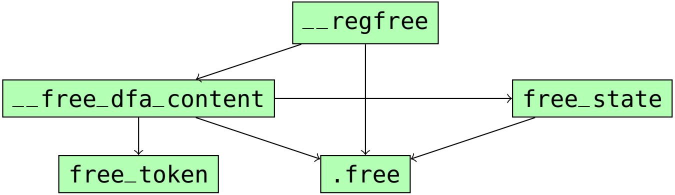
Example:

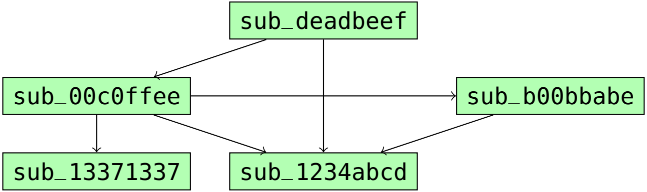

__regfree
__free_dfa_content
free_state
free_token
.free


sub_deadbeef
sub_00c0ffee
sub_b00bbabe
sub_13371337
sub_1234abcd

Technische Universität München

Example:





Semantic feasibility: We define a compatibility relation $\approx$

$$
\begin{aligned}
n \approx m \iff \\
(\forall c \in C_n \, \exists \, c' \in C_m | c = c') \wedge \\
(\forall c \in C_m \, \exists \, c' \in C_n | c = c') \wedge \\
(\forall s \in S_n \, \exists \, s' \in S_m | s = s') \wedge \\
(\forall s \in S_m \, \exists \, s' \in S_n | s = s')
\end{aligned}
$$

This yields the final rule:

$$
\begin{aligned}
F_{sem}(s,n,m) \iff & \\
& n \approx m \land \forall (n',m') \in M(s), (n,n') \in B_1 \Rightarrow (n,n') \approx (m,m') \\
& \land \forall (n',m') \in M(s), (n',n) \in B_1 \Rightarrow (n',n) \approx (m',m)
\end{aligned}
$$

Matching

- Matching is done in brute-force manner
- Multiple sets:
  - Functions that can be exactly identified
  - Functions that have multiple, possible matches
  - Functions that cannot be found via matching (no strings, no constants and no function calls - `IO_default_sync`)

- Test implementation was created
- Free as in Speech (GPLv3 or Later)
- Grab it from github: https://github.com/masterofjellyfish/findmagic
- Disclaimer: You need .NET Framework or Mono
- Supports only x86_64 for now
- Major code cleanup and more architectures (ARM, MIPS) are planned

Exact matches:

definitions	linked against	find-magic	FLIRT (IDA)
`glibc 2.21` (arch linux)	`glibc 2.21` (arch linux)	376	233
`glibc 2.21` (arch linux)	`glibc 2.13` (debian-wheezy)	105	72

- Algorithm can also provide hints
- For example: strcpy_sse2, strcpy_sse3
- Same constants, same callgraph
- Indistinguishable by the algorithm, but they do the same job
- Helpful for manual reversing!

Recovery fails if multiple matching possibilities and function is not part of unique call graph. Example:

```
.CapstoneX86Detail
    push rbp
    mov rbp, rsp
    mov rax, rdi
    add rax, 0x30
    leave
    retn
```

Pseudocode:

```
cs_x86* CapstoneX86Detail(cs_detail *detail) {
    return &detail->x86;
}
```

Questions?

Thanks!