Metasm

Yoann Guillot

11 avril 2017

Metasm

Metasm est un framework opensource
Écrit en ruby, il a pour but :

- de faciliter le travail avec des binaires
- d'assembler
- de désassembler
- de debogguer un/des processus
- de manipuler du code C
- de s'amuser avec l'interpréteur ruby

Plateformes supportées

Plateformes supportées :

- Linux
- Windows
- MacOS (disasm seulement)

Architectures supportées :

- Ia32
- X64
- MIPS
- Sh4, PPC, Z80...

Plan

Structure interne

Différentes classes indépendantes

- ExeFormat (PE, ELF, Shellcode...)
- CPU (Ia32, X64, MIPS...)
- OS

Les fonctionnalités différentes sont dans des fichiers source différents

- encoding/decoding
- Définition d'un cpu
- Définition d'un format d'exécutable

Réutilisation du code

Ruby motto : Dont Repeat Yourself

- Algorithmes génériques décrits dans des objets génériques
- Opérations spécifiques réalisées dans des sous-classes

Exemple :

- ExeFormat peut assembler une séquence d'instruction en une chaîne binaire
- PE/ELF sait comment placer cette chaîne au sein du fichier exécutable

Encapsulation

Les classes reposent l'une sur l'autre pour leurs algorithmes :

- le Disassembler délègue au CPU tout ce qui concerne une Instruction particulière
- le Debugger demande à l'OS de lister les processus

Des classes similaires exposent une API standardisée

CPU	encodage/décodage/interprétation d'instructions
CPU	interface de déboggage (liste des registres, singlestep, hwbp)
ExeFormat	listing des symboles, enumération des sections

Plan

Le désassembleur

Le désassembleur :

- créée et peuple un jeu de DecodedInstructions
- organise et lie les DecodedInstructions en InstructionBlocks
- détecte les Functions
- recense les Xrefs
- gère les labels

Le désassembleur

Il est “intelligent” :

- suit le flot d'exécution
- capable d'une forme d'émulation des instructions : `backtrack`
- gère les sauts calculés simples

Plan

Disassembler

Structure :

- cpu
- program : ExeFormat
- sections : hash adresse $\rightarrow$ section (EncodedData)
- decoded : hash adresse $\rightarrow$ objet (DecodedInstruction, ?)
- function : hash adresse $\rightarrow$ DecodedFunction
- xrefs : hash adresse $\rightarrow$ références vers celle-ci
- prog_binding : hash label $\rightarrow$ adresse

DecodedInstruction

Structure :

- address
- opcode : spécifique au cpu, quelques propriétés génériques (:stopexec, :saveip)
- instruction : notamment la liste des arguments
- block : InstructionBlock
- backtrace_binding : sémantique de l'instruction

InstructionBlock

Structure :

- address
- list : séquence de DecodedInstruction
- to_normal : adresses linkées directement depuis le bloc
- from_normal : adresses arrivant sur ce bloc
- to_subfuncret : addresses linkées depuis ce bloc, quand to_normal pointe sur une sous-fonction (subfuncret pointe sur l'adresse de retour de l'appel)

DecodedFunction

Structure :

- backtrace_binding : sémantique sommaire de la fonction
- backtracked_for : liste d'Expressions à backtracer si on découvre un nouveau chemin menant à la fonction
- return_address : liste des adresses des instructions `ret`

Xref

Structure :

- origin : adresse de l'élément responsable de la xref
- type : type (:x, :r, :w, :address)
- len : taille déréférencée (en bytes) pour les types `:r` ou `:w`

Plan

Utilisation du Disassembler

Création

- depuis un ExeFormat
- exe.disassembler

Désassemblage

- *disassemble*(*address*) : précis mais lent
- *disassemble_fast_deep*(*address*) : beaucoup plus rapide, mais ne va pas résoudre les sauts calculés

disassemble() trick : mettre backtrace_maxblocks_data = -1 avant, pour ignorer les xrefs de données. Fournir les prototypes C pour les librairies externes (utiliser *parse_c*(), et cf samples/factorize-headers-*.rb)

Le désassembleur suit ensuite le code autant que possible.

Utilisation du Disassembler (2)

Récupérer une instruction

- *di_at*(*addr*), renvoie une DecodedInstruction ou nil. Accepte une adresse ou un label.

Récupérer des données brutes

- *read_raw_data*(*addr*, *len*), *decode_dword*(*addr*), *decode_strz*(*addr*)
- ou *get_section_at*(*addr*), qui renvoie la section et son adresse de base

Manipulater les labels

- *set_label_at*(*addr*, *labelname*) / *get_label_at*(*addr*)

Instrumentation du Disassembler

Plusieurs callbacks sont prévus

- *callback_newinstr* : une nouvelle instruction est décodée
- *callback_newaddr* : une nouvelle adresse à désassembler est trouvée
- *callback_finished* : fin du désassemblage

Ils permettent de prendre le contrôle à divers points critiques du processus de désassemblage. Cf les commentaires dans le source pour plus de détails, ou les plugins d'exemple.

Il est aussi possible de redéfinir n'importe quelle fonction utilisées dans le process de désassemblage, via un plugin ou un script.

La GUI

Le désassembleur est livré avec une interface graphique complète.
Elle dépend de :

- GTK2 sur GNU/linux
- l'API native Win32 sous Windows

Elle autorise :

- une vue en listing
- en graphe
- en hexa

On peut ainsi interagir avec le désassembleur.

La GUI (2)

Les commandes principales sont disponibles via le menu.
Principaux raccourcis claviers (similaires a IDA) :

- c : disassemble depuis le curseur
- maj-C : disassemble_fast (fonction courante uniquement)
- ctrl-maj-C : disassemble_fast_deep (fonction et sous-fonctions)
- espace : mode graphe (placer le curseur sur une instruction)
- l/f : listing des labels/fonctions
- x : liste les xrefs vers l'adresse courante
- b : démarre un backtrace à l'adresse courante (exclue)
- ctrl-r : exécution de code arbitraire
- tab : décompilation (très lent, résultats non garantis, x86 uniquement)

Personnaliser la GUI

Les widgets du désassembleur permettent de personnaliser les raccourcis clavier

- keyboard_callback[key] = lambda {}
- keyboard_callback_ctrl[]

On peut également changer la couleur de fond utilisée pour chaque adresse

- bg_color_callback = lambda { |address |rgb_color }

Renvoyer la couleur en RGB, par exemple 'f00' (rouge)

Plan

Deboggage

Deboggage dans Metasm

- Linux/x86
- Windows/x86
- Remote debugging, via GDB serial

Implémentation

Les 3 interfaces sont similaires :

- une cible bas niveau spécifique
- une cible générique utilisant l'api Debugger

La cible générique offre des fonctionnalités avancées

- Breakpoints conditionnels
- Breakpoint avec callback
- Multiprocess/Multithread debugging

Instantiation

- WinDebugger.new(pid) : debug d'un processus existant
- LinDebugger.new(path/to/exe) : creation et debug d'un nouveau processus
- OS.current.find_process('foo').debugger : debug d'un processus existant (by name)
- GdbRemoteDebugger.new("1.2.3.4 :888") : gdb remote debugger sur ce port TCP

Debug interface

L'API consiste principalement en :

- state : récupère l'état de la cible (:running, :stopped, :dead)
- info : plus d'information sur state ("stopped due to signal X", ...)
- get_reg_value(register) : lire la valeur d'un registre du thread actif
- set_reg_value(register, value) : écrit un registre

Debug interface (2)

- bpx(addr, bool_oneshot) { action } : point d'arrêt logiciel, avec callback (optionnel)
- singlestep : relance la cible en mode singlestep
- continue : relance la cible
- check_target : vérifie si un évenement debug est survenu (non bloquant)
- wait_target : comme check_target, bloquant
- resolve() : résoud une expression arithmétique, qui peut référencer des registres, labels et indirections

GUI Debugger

Raccourcis utiles :

- f5 : run
- f10 : stepover
- f11 : singlestep

Dans la console, la plupart des commandes acceptent des expression arbitraires. Elles peuvent faire intervenir ces variables spéciales :

- codeptr : l'adresse du curseur dans la vue 'code'
- dataptr : l'adresse du curseur dans la vue 'data'

eg : Cliquer sur une instruction et taper 'g codeptr' pour l'atteindre.