Élaboration d'une représentation intermédiaire pour l'exécution concolique et le marquage de données sous Windows

Sébastien LECOMTE

cyberaware@laposte.net

SSTIC
06 *juin* 2014

Plan

1. Introduction
2. Représentation intermédiaire
3. Utilisation de PIN
4. Application : FuzzWin
5. Perspectives

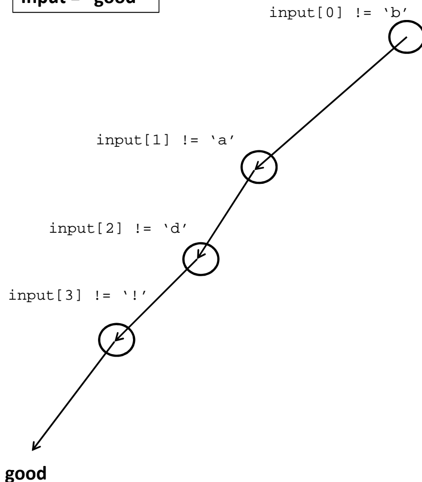
Au commencement...Fuzzgrind

- fuzzer sous architecture GNU/Linux, utilise Valgrind et STP
- écrit par Gabriel Campana et présenté au SSTIC 2009
- basé sur l'algorithme SAGE (Patrice Godefroid - *MS Research*)

- détection d'1/3 des bugs de Windows Seven
- résultats encourageants pour Fuzzgrind en 2009
- adaptons donc Fuzzgrind aux OS Windows ! !

L'algorithme SAGE

Scalable, Automated, Guided Execution

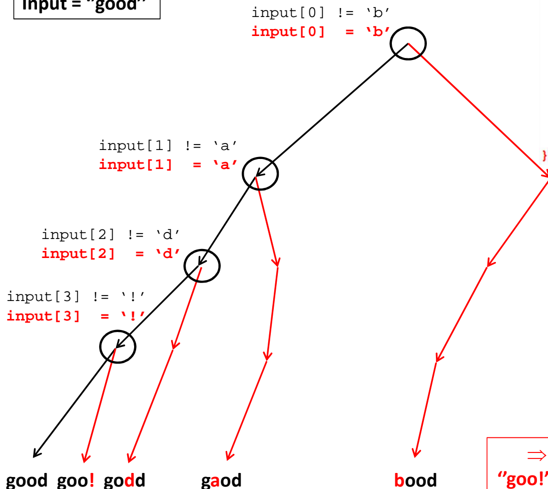



```
void test(char *input)
{
    int cnt = 0;

    /* if (!input) return; */

    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;

    if (cnt > 3) crash();
}
```

L'algorithme SAGE

Scalable, Automated, Guided Execution

Input = "good"

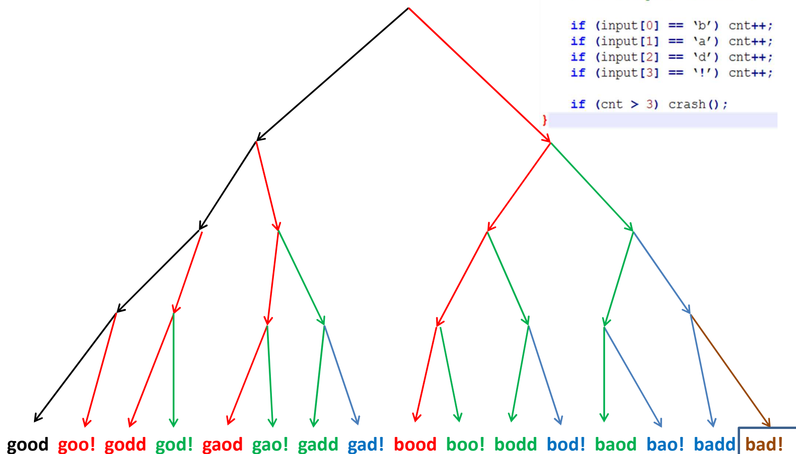



```
void test(char *input)
{
    int cnt = 0;

    /* if (!input) return; */

    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;

    if (cnt > 3) crash();
}
```

⇒ 4 nouvelles entrées : "goo!" "godd" "gaod" "bood"

L'algorithme SAGE

Scalable, Automated, Guided Execution



Le portage vers Windows

OK

- STP $\Longrightarrow$ **Z3** (*Microsoft*)
- Valgrind (Instrumentation) $\Longrightarrow$ **PIN**(*Intel*)

Le portage vers Windows

OK

- STP $\Longrightarrow$ **Z3** (*Microsoft*)
- Valgrind (Instrumentation) $\Longrightarrow$ **PIN**(*Intel*)

KO

- Valgrind (VEX) $\Longrightarrow$? ? ?
- obligation d'écrire une représentation intermédiaire spécifique, adaptée au marquage de données

Plan

1. Introduction
2. Représentation intermédiaire
   - Définitions
   - Création de la représentation intermédiaire
   - Implémentation en C++
   - Exemple : addition 32bits
3. Utilisation de PIN
4. Application : FuzzWin
5. Perspectives

Définition

Marquage de données

Marquage de données - *Data Tainting*

- marque/label sur donnée initiale
- suivi de la propagation au cours de l'exécution
- test du marquage de certaines instructions/branchements
- détection de comportement non souhaité, étude de malware...

Principes

- au moins 1 source marquée => destination(s) marquée(s)
- conservation de l'arithmétique du marquage

Définition

Exécution symbolique

Exécution symbolique

- représentation des instructions et opérandes avec des symboles
- analyse statique ou dynamique
- vérification, détection de fautes, couverture de code. . .

```
mov     eax, 1                    →  vEAX0 = ASSIGN[Value(1,32b)]
shl     eax, 1                    →  vEAX1 = LEFTSHIFT[vEAX0,Value(0, 8b)]
movsx   edx, byte ptr [ecx+eax]   →  vEDX  = SIGNEXT32[ValFromMem(8b)]
```

Définition

Exécution symbolique

Exécution symbolique

- représentation des instructions et opérandes avec des symboles
- analyse statique ou dynamique
- vérification, détection de fautes, couverture de code. . .

```
mov     eax, 1                     →  vEAX0 = ASSIGN[Value(1,32b)]
shl     eax, 1                     →  vEAX1 = LEFTSHIFT[vEAX0,Value(0, 8b)]
movsx   edx, byte ptr [ecx+eax]    →  vEDX  = SIGNEXT32[ValFromMem(8b)]
```

Exécution concolique = concrète + symbolique

Plan

1. Introduction
2. Représentation intermédiaire
   - Définitions
   - Création de la représentation intermédiaire
   - Implémentation en C++
   - Exemple : addition 32bits
3. Utilisation de PIN
4. Application : FuzzWin
5. Perspectives

Représentation intermédiaire

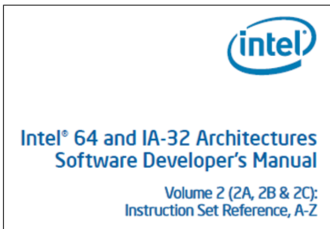
- plusieurs représentations existantes : Miasm, Metasm, REIL...
- le choix a été fait de construire une représentation spécifique
  - -> données dynamiques fournies par PIN
  - -> seules les instructions marquées seront représentées

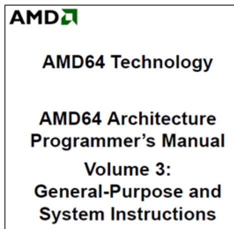
- représentation intermédiaire et non langage intermédiaire
- représentation générique, plusieurs traductions possibles
- 1 traduction implémentée : langage SMT-LIB V2

Représentation intermédiaire

Toutes les instructions suivent le même schéma :

Les sources, destinations et *(presque)* tous les effets sont documentés par les constructeurs





Plan

1. Introduction
2. Représentation intermédiaire
   - Définitions
   - Création de la représentation intermédiaire
   - Implémentation en C++
   - Exemple : addition 32bits
3. Utilisation de PIN
4. Application : FuzzWin
5. Perspectives

Implémentation en C++

Variables symboliques

Classe *TaintObject*

- valeur symbolique d'un emplacement (reg/mem/flag)
- 4 caractéristiques
  - longueur en bits
  - relation qui lie la variable à ses sources
  - sources de la variable
  - nom de la variable lorsqu'elle est traduite

Classe *ObjectSource*

source utilisée dans un *TaintObject*

- si la source est marquée : *TaintObject* associé
- sinon, valeur numérique (*concrète*) de la source

Implémentation en C++

Variables symboliques

```
shl      eax, 1      vEAX1 = LEFTSHIFT[vEAX0,Value(1, 8b)]
```

```
TaintObject vEAX1
_len = 32
_relation = X_SHL
_src = {ObjectSource(vEAX0), ObjectSource(8bits, 1)}
_name = ""
```

Implémentation en C++

Représentation du contexte d'exécution

classes *TaintManager_XX*

- association des *TaintObject* au contexte d'exécution
- si emplacement marqué : pointeur vers *TaintObject*, sinon *nullptr*

Classe *TaintManager_Thread*

- gestion des registres généraux + Flags
- 1 classe par thread d'exécution

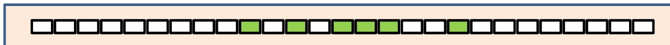
Classe *TaintManager_Global*

- table de hachage pour l'association adresse/*TaintObject*
- crée les premiers objets marqués (lecture fichier en mémoire)

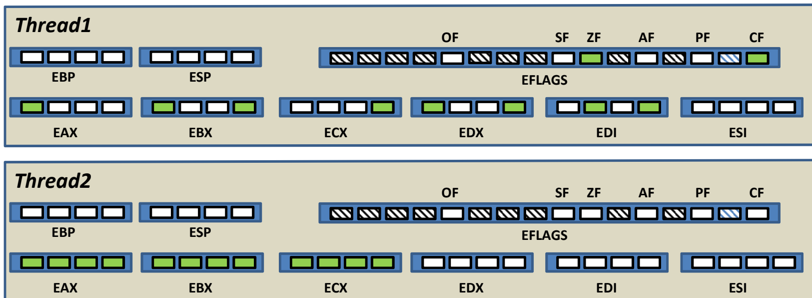
Implémentation en C++

Représentation du contexte d'exécution

TaintManager_Global



TaintManager_Thread



Implémentation en C++

Granularité du marquage

Choix de la granularité

- compromis à trouver entre fidélité et performances
- choix d'un suivi au niveau de **l'octet** (hormis flags)
  - taille minimale d'une opérande x86
  - adapté au marquage initial des données depuis un fichier
- surmarquage principalement dû aux opérations logiques

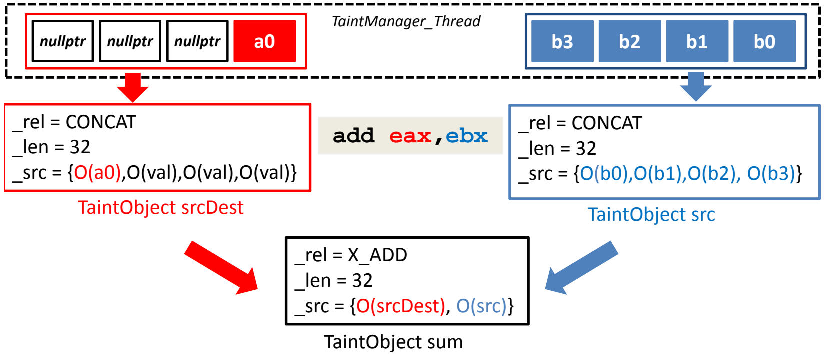
Relations spéciales

- *"CONCAT"* et *"EXTRACT"*
- *"BYTESOURCE"* pour les octets du fichier d'entrée

Plan

1. Introduction
2. Représentation intermédiaire
   - Définitions
   - Création de la représentation intermédiaire
   - Implémentation en C++
   - Exemple : addition 32bits
3. Utilisation de PIN
4. Application : FuzzWin
5. Perspectives

Exemple : addition 32bits

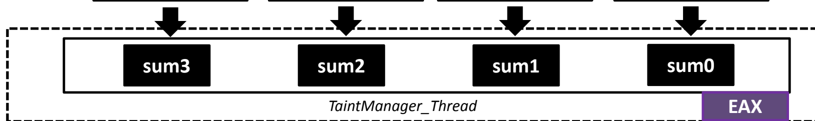


Exemple : addition 32bits

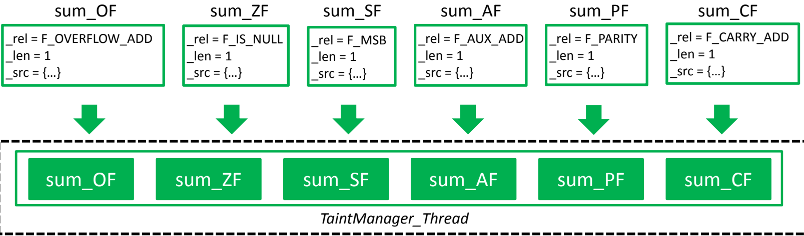
_rel = X_ADD
_len = 32
_src = {O(srcDest), O(src)}

TaintObject sum

TaintObject sum3	TaintObject sum2	TaintObject sum1	TaintObject sum0
_rel = EXTRACT	_rel = EXTRACT	_rel = EXTRACT	_rel = EXTRACT
_len = 8	_len = 8	_len = 8	_len = 8
_src = {O(sum),O(3)}	_src = {O(sum),O(2)}	_src = {O(sum),O(1)}	_src = {O(sum),O(0)}



Exemple : addition 32bits



Plan

1. Introduction
2. Représentation intermédiaire
3. Utilisation de PIN
   - Présentation de PIN
   - Détail de l'instrumentation
4. Application : FuzzWin
5. Perspectives

Présentation de PIN

PIN

- *Framework* d'instrumentation dynamique développé par Intel
- multiplateformes : GNU/Linux, Windows, MacOs, x86/x64
- recompile et exécute le binaire (JIT) après insertions de *hooks* définis par l'utilisateur (*pintool*)
- forum d'utilisateurs et API bien documentée
- cf. présentation de Gal Diskin à HITB 2012

Plan

1. Introduction
2. Représentation intermédiaire
3. Utilisation de PIN
   - Présentation de PIN
   - Détail de l'instrumentation
4. Application : FuzzWin
5. Perspectives

Utilisation de PIN

Détail de l'instrumentation

Instrumentation mise en place

suivi des évènements utiles

- début/fin de chaque *thread* (gestion du contexte associé)
- exécution des appels systèmes (marquage initial)
- exécution de chaque instruction (propagation du marquage)

Utilisation de PIN

instrumentation des appels système

Appels système

- instrumentation avant et après exécution
- suivi des *handles* vers les fichiers suivis (*NtOpenFile, NtCreateFile, NtOpenSection, NtClose*)
- suivi de l'offset de lecture associé (*NtSetInformationFile*)
- marquage de la mémoire après lecture via un *handle* suivi (*NtReadFile*, *NtMapViewOfSection*)

Utilisation de PIN

instrumentation des instructions

Instructions

- ne démarre que lorsque les premières données sont lues
- PIN fournit les informations dynamiques nécessaires (thread, opcode, type de sources et valeur avant ou après exécution, branchement pris ou non, . . .)
- fonction dédiée pour chaque cas (IR, IM, MR, RM, RR)
- traitement de certains cas particuliers
  - opérations logiques avec 0 ou 0xff
  - décalage et rotations multiples de 8 bits
  - idiome de démarquage (xor eax,eax . . .)

Plan

1 Introduction
2 Représentation intermédiaire
3 Utilisation de PIN
4 Application : FuzzWin
5 Perspectives

FuzzWin

Rappel des objectifs

- détecter si les instructions de branchement sont contrôlables
- instructions surveillées : *Jcc, CMOVcc, CALL*

Si branchement marqué. . .

- formule symbolique déterminant le branchement
- inversion de la valeur du branchement à l'exécution
- recherche d'une solution grâce à un solveur de contraintes

FuzzWin

Classe *TranslateIR*

- affectation d'un nom aux variables symboliques utilisées
- traduction des relations entre objets en langage intermédiaire
- traduction des branchements en une formule conditionnelle
- classe abstraite

Classe *TranslateSMTLIB*

- classe dérivée, utilisant le langage SMT-LIB V2

DÉMO ! ! !

Plan

1. Introduction
2. Représentation intermédiaire
3. Utilisation de PIN
4. Application : FuzzWin
5. Perspectives

Perspectives

État de développement

FuzzWin en résumé

- totalement écrit en C++11 (> 20000 *LoC*),
- x86/x64, XP -> Seven, tests en cours sur Windows 8
- toutes les instructions générales sont gérées ($\approx$ 70)
- volonté d'une conception modulaire
  - facilité de portage vers d'autres OS
  - liberté de traduction vers un langage particulier
- Gestion précise du périmètre de test
  - temps maximal d'exécution
  - nombre maximal de contraintes
  - plage d'octets à marquer
  - activation du score des nouveaux fichiers

Perspectives

Travaux en cours

En cours...

- contraintes sur divisions (nullité et débordement)
- adressage indirect des architectures x86
  ex : mov eax, [ebx] $\Longrightarrow$ *quid* si ebx marqué ? ?
- amélioration de la finesse d'exécution
  - marquage limité à une bibliothèque
  - marquage limité à une fonction spécifique du binaire
- Enrichissement du langage
  - instructions SIMD sur les entiers (MMX, SSE...)
  - instructions sur les réels

Perspectives

Évolutions possibles

Parmi les idées...

- méthode de gestion des boucles (analyse statique ?)
- détection *Heap/stack overflow, UAF*. . .

  $\Longrightarrow$ blog de Jonathan Salwan (www.shell-storm.org)
- implémentation autre traduction (langage C)
- instrumentation en *ring0*

Merci de votre attention ! !