Désobfuscation automatique de binaires

Et autres idyles bucoliques...

Alexandre Gazet
Sogeti / ESEC R&D
alexandre.gazet(at)sogeti.com

Yoann Guillot
Sogeti / ESEC R&D
yoann.guillot(at)sogeti.com

SSTIC 2009

Plan

Point de départ

Nécessité d'automatisation

- Nous avons déjà :
  - Processeur filtrant
  - Parcours du graphe de contrôle (`CFG`)
  - Application de règles de réécriture
  - Modification à la volée du `CFG`

Analyse manuelle du code obfusqué

- Recherche manuelle de motifs
  - Pénible
  - Peu générique
  - Éventuellement inefficace : ex résistance au polymorphisme

Nouvelle approche

Besoins

- Conservation de la sémantique
- Réécriture du code dans une forme plus simple
- Élimination du code mort
- etc.

- Les compilateurs le font déjà : **l'optimisation**
- Notre critère d'optimisation : la concision du code

Présentation de la cible

La cible

- Une protection à base de virtualisation
- Massivement obfusquée (difficulté d'analyse + polymorphisme)

Approche proposée

- Greffe d'un module d'optimisation sur le module du parcours de graphes

Constant propagation

```
cfh mov al, 12h
67h mov cl, 46h
69h xor cl, al
```

```
cfh mov al, 12h
67h mov cl, 46h
69h xor cl, 12h
```

FIG.: Propagation de 12h à travers *al*.

Constant folding

```
cfh  mov  al ,  12h
67h  mov  cl ,  46h
69h  xor  cl ,  12h
```

```
cfh  mov  al ,  12h
67h  mov  cl ,  54h
```

FIG.: Réduction de *cl*.

Operation folding

```
4fh add al, -7fh
51h add al, bl
53h add al, -70h
```

```
4fh add al, 11h
51h add al, bl
```

FIG.: Réduction de l'opération *add*.

Démo

Optimisation d'un handler

Analyse sémantique des handlers

- Méthode `code_binding` de l'objet `disassembler`

Exemple de handler optimisé

```
lodsd
xor eax, ebx
add eax, 859fcfaeh
sub ebx, eax
push eax
```

- Sémantique (*binding*)

```
dword ptr [esp] := (dword ptr [esi]^ebx)+859fcfaeh
eax := (dword ptr [esi]^ebx)+859fcfaeh
ebx := ebx+-(dword ptr [esi]^ebx)-859fcfaeh
esi := esi+4
esp := esp-4
```

Au commencement

2ème projection de Futamura

Étant donnnés deux langages L_a et L_b, il est possible de trouver un compilateur de L_b vers L_a, si on connaît un interpréteur de L_b écrit en L_a

Comment ?

Du statique au (presque) dynamique

- Binding contextualisé :

```
dword ptr [esp] := 0c0000001h
eax := 0c0000001h
ebx := 4000fd8ch
esi := 100167c2h
esp := esp-4
```

- Assembleur généré : `push 0c0000001h`
- Suivi du flot d'exécution du bytecode

Démo

Exécution symbolique et génération assembleur

Interprétation du résultat

- L'intégralité du *bytecode* a été compilée en asm `Ia32` natif
- Des références au contexte de l'interpréteur
- Proche d'un automate à pile

$\Rightarrow$ **module d'optimisation + abstraction**

Injection d'abstraction

Extension du processeur

```
la32::Reg.i_to_s[32].concat( %w[virt_eax])
la32::Reg.i_to_s[16].concat( %w[virt_ax])
la32::Reg.i_to_s[8].concat( %w[virt_al])

la32::Reg.s_to_i.clear
la32::Reg.i_to_s.each { |sz, hh|
    hh.each_with_index { |r, i|
        la32::Reg.s_to_i[r] = [i, sz]
    }
}
la32::Reg::Sym.replace la32::Reg.i_to_s[32].map {|s|s.to_sym}
```

Démo

Chunk optimisé avec registres virtuels

Phase finale

- Injections des registres virtuels
- Optimisation
- Expression du code en registres virtuels uniquement
- Registres virtuels *remappés* sur les registres natifs
- Compilation et édition des liens

Démo

Code dévirtualisé, mappé dans le binaire original

Conclusion(s) 1/2

- **Méthodes d'optimisation** (règles de réécriture)
  - Très efficaces
  - Implementation limitée
    - localité des optimisations
    - manque d'une représentation intermédiaire
    - inadaptée aux obfuscations du flot de contrôle

- **L'évaluation partielle ou spécialisation**
  - Pré-calcul de tous les éléments statiques :
    - Mouvements de données dans le code obfusqué
    - Application de l'interpréteur au *bytecode*
  - Approche générique
  - Relativement coûteux en temps de calcul

Conclusion(s) 2/2

- **Intégration et réutilisation du désobfuscateur**
  - Le code actuel est à l'état de prototype
  - En cours d'intégration au framework sous forme d'un plugin propre
  - Utilisable sur tout code x86, avec des parties cross-platforme

Plan

Décompilation

- Interface arch-spécifique réduite
- Meilleure expressivité du code
- Représentation plus simple des boucles
- Sémantique plus simple à manipuler

Décompilation : limitations

- Des constructions assembleur sont difficiles à transcrire en `C`
  - *rol*, *ror*
  - *jmp eax*
- Nécessité que le code soit formatté correctement
  - Fonction/sous-fonctions
  - ABIs/Conventions d'appels
- On peut imaginer des moyens de s'en accomoder

Démo

Décompilation

Plan

N'oubliez pas

- **Metasm**
  `https://metasm.cr0.org/`
- **Blog**
  `http://esec.fr.sogeti.com/blog/`

Conclusion

Merci pour votre attention.

Des questions ?