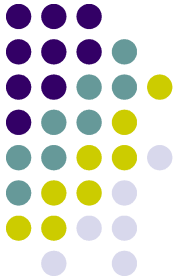
Travail d'étude
Les techniques de protection du logiciel

Présenté par:

Julien **Burle** Térence **Scalabre**

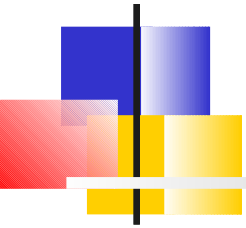
Licence Informatique

Comment et pourquoi protéger son logiciel ?



- Tout d'abord intéressons-nous aux raisons de protéger son code. Nous traiterons aussi de l'aspect juridique de la défense d'un logiciel.

- Puis nous étudierons les techniques pratiques de protections appliquées au langage C sous l'environnement UNIX.



Pourquoi protéger son programme ?

Les avantages et inconvénients de la protection:

Avantages :

- Empêcher la réutilisation des techniques développées
- Contrôler les versions du client
- Ne pas dévoiler d'éventuelles failles

Inconvénients :

- Impossibilité d'amélioration par un tiers
- Difficulté de réutilisation et de lisibilité du code
- Coût des protections en temps et en argent

Aspects juridiques :

La loi française :

- Protection contre le piratage et l'espionnage industriel.
- Droit Commun: interdiction de pénétrer dans un système élément de la propriété d'autrui.
- Droit d'Auteur: le logiciel est considéré comme une oeuvre de l'esprit
- La décompilation est tolérée sous certaines conditions:
  - droit d'utilisation du logiciel
  - décompilation limitée au nécessaire
  - on ne peut pas écrire un nouveau logiciel à partir d'un logiciel décompilé

  Les informations acquises ne doivent servir qu'à la réalisation de l'interopérabilité.

Aspects juridiques (suite):

La loi européenne :

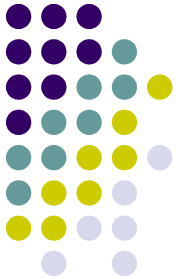
Le conseil européen a proposé le 18 mai 2004, une directive sur la brevetabilité des inventions mises en oeuvre par ordinateur, telles que les algorithmes, types de fichiers ou logiciels.

Cette décision pourrait contraindre les acteurs du logiciel libre à des dépôts de brevets défensifs.

Cependant les techniques de protections sont couramment utilisées.

Étudions leurs principes de fonctionnement.

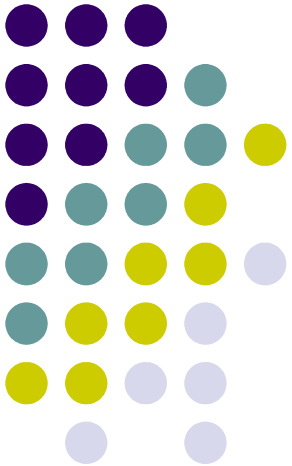
Techniques de protections du code source

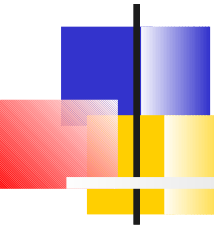


Nous pouvons distinguer plusieurs types de protections:

- Celles visant à empêcher l'utilisation d'un débogueur.
- Celles n'autorisant l'exécution du code que sous certaines conditions
- Celles visant à rendre le code désassemblé incompréhensible.

Techniques anti-débogueur





Détecter l’appel système **ptrace**

Détecter l'appel systéme ptrace

- L'appel système ptrace sous UNIX permet de tracer un processus (changer sa mémoire, modifier ses registres...). Il sert aussi à déboguer les processus.

➢ Comment l'utiliser comme protection?

Un processus peut être tracé qu'une fois.

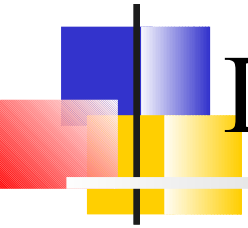
Grâce à la requête TRACEME, un programme se trace lui-même, ce qui empêche une autre trace ou détecte une trace existante.

Détecter l'appel systéme ptrace

- Exemple d'utilisation :

```
int main(void) {
   if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0) {
            printf("DEBUGGING... Bye\n");
            return 1;
   }
   printf("Hello\n");
   return 0;
}
```

- On peut également appliquer cette technique à strace qui a un fonctionnement similaire.



Détecter les points d’arrêt

Détecter les points d'arrêt (breakpoints)

- Un point d'arrêt (breakpoint) permet dans les débogueurs de faire une pause ou de stopper l'exécution d'un processus.

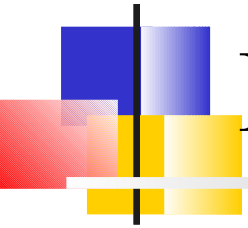
➢ Comment détecter un breakpoint?

- L'instruction assembleur d'un breakpoint est int3 (l'opcode correspondant est 0xCC).
- Lorsqu'un débogueur met un breakpoint à une certaine adresse il sauve l'opcode courant pour mettre à la place 0xCC.

Détecter les points d'arrêt (breakpoints)

- Pour se protéger des breakpoints, on va scanner notre code à la recherche de l'opcode 0xCC.

- Seul désavantage de cette technique, un code peut contenir l'opcode 0xCC sans que ce soit forcément un breakpoint.



Poser de faux breakpoints

Poser de faux breakpoints

- Comme la précédente, cette méthode s'appuie sur la détection des breakpoints.
- Lorsqu'un programme rencontre l'opcode 0xCC, il reçoit un signal SIGTRAP.

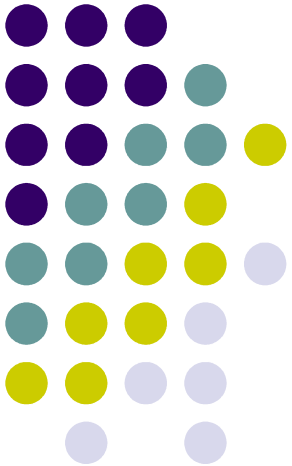
➢ Comment modifier le comportement par défaut ?

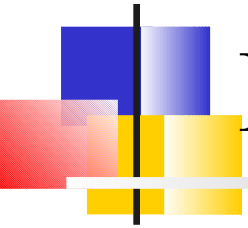
- En utilisant la fonction signal nous allons traiter le signal SIGTRAP.

Considérons par exemple ce programme :

```
void sighandler(int signal) {
   printf("Le pass est \"salut\"\n");
   exit(0);
}
int main(void) {
   signal(SIGTRAP,sighandler);
   __asm__("int3");
   printf("pris au piege\n");
   return EXIT_FAILURE;
}
```

Protection de l'exécution





Protection par code Checksum

Protection par code Checksum

- Cette méthode vérifie si le code original n'a pas été modifié.
- A chaque exécution du programme, un algorithme vérifie que la somme des opcodes du programme n'est pas changée.

Etudions le programme suivant :

```
globl main
main:
        xorl %eax,%eax
        cmp $1,%al
        je coool

pascoool:
/* exit(0); */

exit:
        xorl %eax,%eax
        xorl %ebx,%ebx
        inc %eax
        int $0x80
```

```
coool:
        xorl %eax,%eax
        movb $4, %al            // 4 = SYS_write
        xorl %ebx,%ebx
        inc %ebx                // 1 = STDOUT
        movl $chaine,%ecx  // l adresse de Gagne
        xorl %edx,%edx
        mov $8,%edx             // on ecrit 8 octets
        int $0x80       // on execute le syscall write
jmp exit

chaine:
.string “Gagne !\n”
```

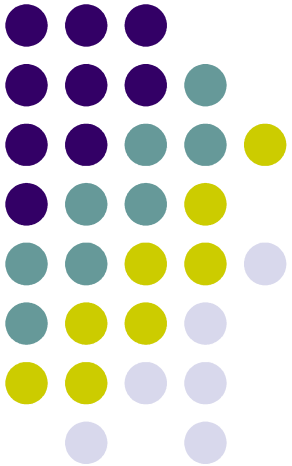
Voyons comment protéger la partie coool en implémentant un checksum.

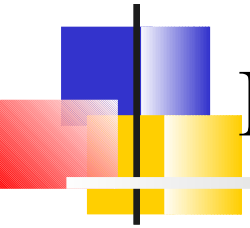
Nous ajoutons le test de vérification :

```
.globl main
main:
        jmp checksum
rett:
        xorl %eax,%eax
        cmp $1,%a1
        je coool
pascoool:
exit:
        [...]
coool:
        [...]
        jmp exited
```

```
checksum:
        xorl %ebx,%ebx
        mov $checksum,%ecx
        sub $main,%ecx
        // boucle de main jusqu'àchecksum
        mov $main,%esi        // on part de main
boucle:
        lodsb
        add %eax,%ebx // la somme dans ebx
        loop boucle
        cmpl $4079,%ebx
        jne baaad          // si la somme est fausse
        jmp rett
baaad:
// message d'erreur
[...]
jmp exit
[...]
```

Techniques anti-désassemblage





Faux désassemblage (false dissasembly)

- Le but est de brouiller le code désassemblé.

- Comment fausser les résultats ?
  - On rajoute dans notre code, les opcodes d'un jump vers une instruction que nous voulons exécuter.
  - Entre ces deux parties, on place de faux opcodes destinés à faire perdre l'alignement du code désassemblé

- Voici un exemple de code ainsi traité :

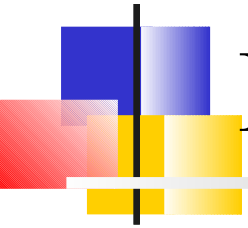
```
main:
    xorl %eax,%eax
    movb $3,%al
    xorl %ebx,%ebx
    movl %esp,%ecx
    mov $4, %edx
    int $0x80
.ascii "\xEB\x01\xE8"
    cmpl $0x73736170, (%ecx)
    je good
[…]
```

Faux désassemblage (false dissasembly)

- La ligne .ascii "\xEB\x01\xE8" ajoute les opcodes d'un jmp +1 (\xEB\x01), puis un opcode de début de call (\xE8) (que l'on va sauter).

- L'alignement est donc perdu au désassemblage.

- Cette technique est donc très utile pour cacher une portion sensible de code (par exemple lorsque l'on détecte l'appel système ptrace).



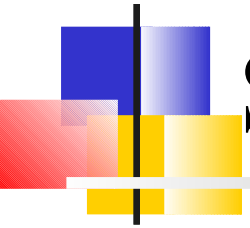
Polymorphisme

- Le but est de décourager le reverser
  - Pour arriver à un résultat voulu nous allons passer par de nombreuses étapes inutiles.

- Par exemple :

L'instruction `movl $1,%eax` peut être remplacée par

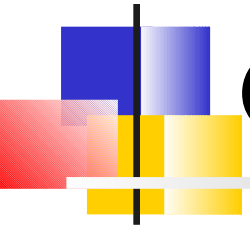
```
xorl %eax,%eax
cdq              // équivaut à un xor %edx,%edx
leal 1(%edx),%eax  // place 1 dans eax
movl %fs,%ebx  // le registre fs contient toujours 0
add %ebx,%eax  // équivaut à add $0,%eax
```



Strip

Strip

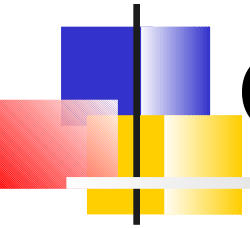
- Cette technique utilise l'outil STRIP sous UNIX.
  - Cet outil enlève toutes les références aux symboles dans un binaire (sans changer le fonctionnement du programme).
  - Au désassemblage, un binaire traité par STRIP ne contient plus de références aux noms des fonctions (même pas le main).
  - Il devient plus difficile de mettre un point d'arrêt sur une fonction que l'on doit alors désigner par son adresse.



Cryptage

Cryptage

- Une méthode célèbre pour compliquer l'ingénierie inverse consiste à crypter la section TEXT (le code) directement dans le binaire.
- Au démarrage, un bout de code va décrypter toute la section afin que le programme s'exécute normalement.



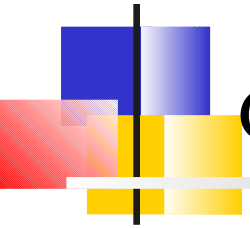
Obfuscateurs

Obfuscateurs

- Cette méthode complique le code source d'un programme afin de le rendre le moins compréhensible possible.

- Cette technique n'est malgré tout pas très puissante. Il suffit d'un peu de temps et de courage pour retrouver le code initial.

- Elle pourra retarder le reverser déterminé ou décourager les novices.

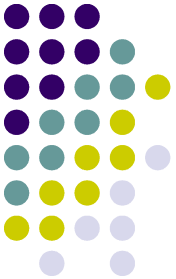


Conseils divers lors de l’implémentation de protection

Conseils divers lors de l'implémentation de protection

- Ne jamais laisser un mot de passe codé en clair.
- Pas de message d'erreur
- Protection agressive (désactivation du clavier, plantage de la machine)
- Utilisation de saut conditionnel

Pour conclure



- Utilisées de façon combiné ces techniques simples permettent des protections plus complexes.
- L'implémentation de ces techniques est cependant très coûteuse en temps et en argent, sans être infaillible.

➢ Après cette étude nous pensons qu'un programmeur n'a pas intérêt de protéger son logiciel (avantages : correction de bugs, possibilité d'amélioration et d'évolution, gain de temps…).