Organisation

- mardi 03.06.2014 TEP2
  Cours : Protections
- mardi 10.06.2014 Labo 5 - Lizord
- mardi 17.06.2014 TEP3 (exploitations, protections) ???
  Cours : Protections et exemple
- mardi 24.06.2014 Labo 6 - Tweeter (avec la faute d'orthographe)
- mardi 08.07.2014 EXAMEN
  9h00 - 11h00 en G01
  Résumé personnel : 1 feuille A4 (recto-verso)
  Aide-mémoire (SLO14_slides3b_aide_memoire.pdf)

SLO

Sécurité logicielle

Chapitre 6.
Protections logicielles

Sylvain Pasini
sylvain.pasini@heig-vd.ch
Mars - Juillet 2014

heig-vd Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Aperçu

Aperçu

Comment les failles sont découvertes ?

- La communauté recherche activement des failles :
  - étude du code binaire
  - si possible, étude du code source
- Les chercheurs sont souvent
  - les développeurs eux-mêmes
  - des équipes dédiées (validation, qualité, sécurité)
  - des auditeurs externes (experts, éditeurs de logiciels)
  - des pirates
- Si une faille devient publique, il est nécessaire de la patcher
  - les pirates essaient de garder des failles exploitables (0-day)

Les vulnérabilités

- buffers overflows
- integer overflows
- format strings
- mais aussi les meta-caractères :
  - injection SQL
  - cross-site scripting
  - etc.
- side-channels (info leak, timing attacks, etc.)
- bug et problèmes de logique
- et bien d'autres.

Eviter les vulnérabilités

- Programmer correctement ;-)
  - démarrer le projet en pensant à la sécurité (très rare)
  - respecter les security guidelines (code, standard, ...)
  - vérifier tous les cas possibles (negative-size bug, integer overflow, ...)
  - éviter toute erreur sur la taille d'un buffer (buffer overflows)
  - éviter d'écrire hors des zones allouées (buffer overflows)
  - éviter d'écrire avec des caractères inattendus (format string, ...)
  - vérifier les types et leur interprétation (signé, non-signé, integer overflow, ...)
  - etc.
- Bonne chance !

De manière plus réaliste

- Pratiquer la revue de code (code-review).
  - le code doit être relu par une (ou plusieurs) personne(s)
  - pas par le programmeur lui-même
- Rendre l'exploitation très difficile

En pratique

- Souvent, le management voit la sécurité comme "secondaire"
  - ce n'est pas le "core business",
  - et même si ça l'est, l'aspect fonctionnel est prioritaire
- Les programmeurs ne sont souvent pas des experts en sécurité
  - d'où la nécessité d'une équipe sécurité qui fait de la revue de code
- Les produits deviennent de plus en plus complexes
  - difficile de penser à la sécurité en plus du reste
- Les programmeurs doivent être efficaces
  - pas le temps de traîner pour écrire le code
  - pas le temps pour la revue de code (ou partielle)
- Avec tout cela, les erreurs sécuritaires arrivent plus vite qu'on ne le pense

Au final

- Les produits sortent sur le marché avec des failles
  - la sécurité n'est peut-être pas prise en compte dès le début du développement, cela coûtait trop cher de la rajouter en cours de développement
- Les vulnérabilités identifiées (par les pirates ?) seront patchées.
  - quel en est le coût ? (coût de la brèche, coût du patch, ...)
- Au fur et à mesure des versions, les vulnérabilités diminueront
  - du moins, on l'espère...
- La sécurité commence a être prise en compte dès le départ
  - Bill Gates : "Security is our biggest challenge ever"

Audit de logiciels

- Inspection manuelle
  - inspection du code source par un auditeur
  - dépend beaucoup des compétences de l'auditeur
  - dépend beaucoup du temps à disposition

Audit de logiciels

- Inspection automatisée
  - Analyse de code source
    - peu efficace, seules les grosses failles sont découvertes
    - exemple : RATS, LINT
  - Analyse de code binaire
    - utilisation de désassembleurs (statique, similaire à l'analyse de code source)
    - utilisation de débugueurs (dynamique)
    - exemple : IDA, Nessus, OllyDBg, SoftIce
  - Profilers
    - normalement utilisé pour identifier des fuites mémoire
    - permet de suivre les opérations mémoires
    - exemple : Rationale Purify, Valgrind
  - Fuzzers
    - permet de découvrir des failles complexes
    - nécessite la compréhension du format d'entrée pour être efficace
    - exemple : Autodafé, Spike, Protos

Recherche de vulnérabilités

- Les méthodes sont souvent
  - couteuses
  - peu efficaces
  - difficiles à mettre en oeuvre
- Il est nécessaire d'éviter au maximum les vulnérabilités
  - cela inclut le programmeur, le compilateur, etc.
- Au moins, **limiter l'impact** des vulnérabilités
- Nous allons voir quelques techniques permettant de limiter leur impact

Aperçu

Pile non-exécutable

- Protection directe contre le problème de "stack smashing"
- Idée principale :
  - **une instruction située à une adresse appartenant à la pile (stack) ne peut pas être exécutée**
  - donc, si EIP pointe sur une adresse de la pile (typiquement sur un shellcode), l'exécution est impossible
- Utilisation du bit NX (No eXecute, x86 et x86-64)
  - Supporté par les processeurs AMD, Intel, ...
  - Si le hardware ne le supporte pas, le bit NX est parfois émulé.
- Avantage du patch :
  - pas de perte de performance
  - pas de re-compilation nécessaire
    - pour le fonctionnement du programme
    - pour la protection du programme (à voir en pratique…)
- Inconvénient :
  - noyau (Linux) compilé spécialement avec ce patch

Pile non-exécutable

- Patch pas évident...
  1. GCC utilise la stack exécutable... aïe !
     - Elle est utilisée pour des "trampolines" avec des fonctions imbriquées. Du code exécutable est créé "on-the-fly" sur la pile.
  2. Linux utilise la stack exécutable pour le "signal handling".
  3. Quelques langages/programmes nécessitent la stack exécutable
- Le patch gère tous ces problèmes.
- Pour 1. et 3. :
  - Il détecte ces cas précis.
  - Il autorise (de manière permanente) la stack exécutable pour ce processus.
- Pour 2. :
  - Il autorise la stack exécutable le temps du signal.

Pile non-exécutable : faiblesses

- Les compromis du patch permettent de potentielles intrusions
  - Par exemple :
    - un buffer overflow dans un signal handler
- Il se peut que l'attaque sur la pile mette le code exécutable ailleurs :
  - Par exemple :
    - dans buffer statique (hors de la stack)
    - sur le tas
- Il se peut que l'attaque manipule la pile sans ajouter de code
  - Par exemple :
    - le code nécessaire est disponible (return to libc)
    - modification de l'adresse de retour pour sauter un morceau de code

Tas non-exécutable

- Même principe que la pile non-exécutable.

Aperçu

Le terme « canari »

- Il s'agit de canaris (oiseaux) qui ont été utilisés dans les mines de charbon.
- Ces canaris étaient utilisés pour détecter les gaz toxiques tels que le monoxyde de carbone.
- L'idée était que le canari meurt de ce gaz avant que les mineurs ne meurent.
- Ca permettait aux mineurs de s'échapper avant qu'ils aient été tués avec le canari.

Canaris (stack guard)

- Publication originale :
  - **StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks**
  - Usenix Security Symposium 1998
  - Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang

Canaris (stack guard)

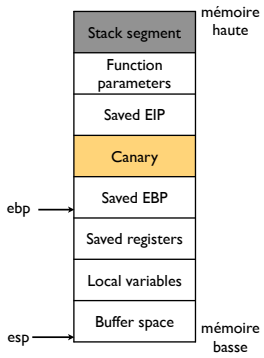
- Valeur par défaut
- Prologue

```
move canary-index-constant into register[5]
push canary-vector[register[5]]
```

- Epilogue

```
move canary-index-constant into register[4]
move canary-vector[register[4]] into register[4]
exclusive-or register[4] with top-of-stack
jump-if-not-zero to constant address .canary-death-handler
add 4 to stack-pointer
< normal return instructions here>
.canary-death-handler:
```

- Avantage :
  - Stoppe les exploits actuels
- Inconvénient :
  - Les exploits sont « facilement » adaptables



Canaris (stack guard)

- Type "Terminator"
  - Valeur constante par défaut : 00, CR, LF, -1
  - Valeur connue par l'adversaire
- Type "Random"
  - Choix de valeurs aléatoires au lancement du programme
    - une valeur différente par canari
- Type "Random XORed"
  - Le canari aléatoire est "xor-é" avec la valeur protégée
  - De cette manière, les deux dépendent l'une de l'autre

Aperçu

Randomisation de la mémoire

- Rappel sur l'espace d'adressage :
  - Chaque processus possède son propre espace d'adressage virtuel.
- Exploiter une vulnérabilité :
  - Pour dérouter un programme, il est nécessaire de connaître l'adresse cible
- Contre-mesure :
  - Rendre difficile la prédiction d'adresses
  - Arrangement aléatoire des adresses « importantes »
  - ASLR : Address Space Layout Randomization
  - Plages d'adresses concernées :
    - code exécutable
    - positions des librairies (diminue le succès des return to libc)
    - pile (stack)
    - tas (heap)

Randomisation de la mémoire

- Linux
  - natif depuis kernel 2.6.12 (juin 2005), faible ASLR...
  - existe des patchs "meilleurs" comme PaX ou ExecShield
- Microsoft Windows
  - natif depuis Vista (janvier 2007)
    - uniquement pour les logiciels linkés pour du ASLR-enabled (pas le cas d'IE7...)
- Apple Mac OS
  - natif depuis Mac OS 10.5 Leopard (octobre 2007)
- Apple iOS
  - natif depuis 4.3

ASLR : Activation/désactivation

- Linux (privilèges root) :
  - Pour le désactiver :
    - echo 0 > /proc/sys/kernel/randomize_va_space
  - Pour l'activer
    - echo 1 > /proc/sys/kernel/randomize_va_space
- Windows :
  - Il peut être désactivé à l'aide de « Enhanced Mitigation Experience Toolkit »
    source : http://www.microsoft.com/en-us/download/details.aspx?id=41138

Aperçu

Aperçu

- StackGuard
- Stack Smashing Protector (SSP), ou ProPolice
- ExecShield
- PaX
- Diverses options de GCC

GCC : StackGuard (1997)

- Extension pour GCC
  - Protège contre les attaques par débordement de tampon sur la pile (surtout stack smash)
  - Protège la sauvegarde de EIP (backup EIP sur la pile)
  - Fournit une protection de type canari
- Implémentation
  - Initialement avec un canari de « 0 »
  - Puis, avec Terminator-, Random-, et Random-XOR-canary
- Jamais accepté comme standard dans GCC 3.x...

GCC : Stack-Smashing Protector (SSP, 2005)

- Aussi connu sous le nom de "ProPolice"
- C'est une amélioration de StackGuard
  - protection de tous les registres sauvés sur la pile (StackGuard protège uniquement EIP)
  - protège les arguments en créant une copie
  - d'autres "petites" modifications
- Natif dès GCC 4.1 (patch pour GCC 3.x)
- Activation/désactivation
  - -fstack-protector/-fnostack-protector (stack-smahing protection)
  - -fstack-protector-all (toutes les protections)

- Page officielle : http://www.research.ibm.com/trl/projects/security/ssp/

SSP : Page MAN de GCC

-fstack-protector

Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call alloca, and functions with buffers larger than 8 bytes.

The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits.

NOTE: In Ubuntu 6.10 and later versions this option is enabled by default for C, C++, ObjC, ObjC++, if neither -fno-stack-protector nor -nostdlib are found.

-fstack-protector-all

Like -fstack-protector except that all functions are protected.

SSP : Code assembleur (printme_bounded.c)

```
08048484 <vulfunc>:
 8048484:  55                  push   %ebp
 8048485:  89 e5               mov    %esp,%ebp
 8048487:  83 ec 38            sub    $0x38,%esp
 804848a:  8b 45 08            mov    0x8(%ebp),%eax
 804848d:  89 44 24 04         mov    %eax,0x4(%esp)
 8048491:  8d 45 d8            lea    -0x28(%ebp),%eax
 8048494:  89 04 24            mov    %eax,(%esp)
 8048497:  e8 fc fe ff ff      call   8048398 <strcpy@plt>
 804849c:  8d 45 d8            lea    -0x28(%ebp),%eax
 804849f:  89 04 24            mov    %eax,(%esp)
 80484a2:  e8 01 ff ff ff      call   80483a8 <puts@plt>
 80484a7:  b8 00 00 00 00      mov    $0x0,%eax
 80484ac:  c9                  leave
 80484ad:  c3                  ret
```

```
080484d4 <vulfunc>:
 80484d4:  55                  push   %ebp
 80484d5:  89 e5               mov    %esp,%ebp
 80484d7:  83 ec 58            sub    $0x58,%esp
 80484da:  8b 45 08            mov    0x8(%ebp),%eax
 80484dd:  89 45 c4            mov    %eax,-0x3c(%ebp)
 80484e0:  65 a1 14 00 00 00   mov    %gs:0x14,%eax
 80484e6:  89 45 f4            mov    %eax,-0xc(%ebp)
 80484e9:  31 c0               xor    %eax,%eax
 80484eb:  8b 45 c4            mov    -0x3c(%ebp),%eax
 80484ee:  89 44 24 04         mov    %eax,0x4(%esp)
 80484f2:  8d 45 d4            lea    -0x2c(%ebp),%eax
 80484f5:  89 04 24            mov    %eax,(%esp)
 80484f8:  e8 e3 fe ff ff      call   80483e0 <strcpy@plt>
 80484fd:  8d 45 d4            lea    -0x2c(%ebp),%eax
 8048500:  89 04 24            mov    %eax,(%esp)
 8048503:  e8 f8 fe ff ff      call   8048400 <puts@plt>
 8048508:  b8 00 00 00 00      mov    $0x0,%eax
 804850d:  8b 55 f4            mov    -0xc(%ebp),%edx
 8048510:  65 33 15 14 00 00 00 xor   %gs:0x14,%edx
 8048517:  74 05               je     804851e <vulfunc+0x4a>
 8048519:  e8 d2 fe ff ff      call   80483f0 <__stack_chk_fail@plt>
 804851e:  c9                  leave
 804851f:  c3                  ret
```


SSP : Comportement

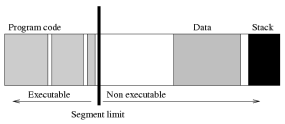
- Sans SSP, on aura probablement :
  Segmentation fault
- Avec SSP, on aura :

```
*** stack smashing detected ***: ./passwd_checker_stack_prot terminated
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x50)[0xb7f562d0]
/lib/tls/i686/cmov/libc.so.6(+0xe227a)[0xb7f5627a]
./passwd_checker_stack_prot[0x8048587]
[0x36353433]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 271595     /root/HEIG/SSP12/labo2_stack_overflow/passwd_checker_stack_prot
08049000-0804a000 r--p 00000000 08:01 271595     /root/HEIG/SSP12/labo2_stack_overflow/passwd_checker_stack_prot
0804a000-0804b000 rw-p 00001000 08:01 271595     /root/HEIG/SSP12/labo2_stack_overflow/passwd_checker_stack_prot
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7e54000-b7e71000 r-xp 00000000 08:01 788956     /lib/libgcc_s.so.1
b7e71000-b7e72000 r--p 0001c000 08:01 788956     /lib/libgcc_s.so.1
b7e72000-b7e73000 rw-p 0001d000 08:01 788956     /lib/libgcc_s.so.1
b7e73000-b7e74000 rw-p 00000000 00:00 0
b7e74000-b7fc7000 r-xp 00000000 08:01 920678     /lib/tls/i686/cmov/libc-2.11.1.so
b7fc7000-b7fc9000 r--p 00153000 08:01 920678     /lib/tls/i686/cmov/libc-2.11.1.so
b7fc9000-b7fca000 rw-p 00155000 08:01 920678     /lib/tls/i686/cmov/libc-2.11.1.so
b7fca000-b7fcd000 rw-p 00000000 00:00 0
b7fe0000-b7fe2000 rw-p 00000000 00:00 0
b7fe2000-b7fe3000 r-xp 00000000 00:00 0          [vdso]
b7fe3000-b7ffe000 r-xp 00000000 08:01 788898     /lib/ld-2.11.1.so
b7ffe000-b7fff000 r--p 0001a000 08:01 788898     /lib/ld-2.11.1.so
b7fff000-b8000000 rw-p 0001b000 08:01 788898     /lib/ld-2.11.1.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
Aborted
```


Exec Shield

- Projet démarré par RedHat en 2002
- Objectif : réduire la propagation automatique de maliciels (Linux)
  - conséquences des vers SQL slammer (Win) et CodeRed (Win),
  - puis du ver Slapper (Linux 2002).
- Principe :
  - Emulation du bit NX pour les architectures sans support HW
- Implémentation du "Segment limit"
  - mémoire de donnée : flag non-exécutable
  - mémoire d'instruction : flag non-modifiable

Mémoire virtuelle du processus
(choix de la limite par le noyau)



source : http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf

Exec Shield : PIE

- Avec Linux,
  - Un programme est normalement compilé pour fonctionner à une adresse précise
  - Pas de randomisation possible pour l'adresse du code
  - Cela est une vulnérabilité...
- PIE pour Position Independent Executable
  - Intégré à GCC
  - Permet de compiler un programme de manière à ce qu'il fonctionne à n'importe quelle adresse mémoire
  - Si le noyau supporte le PIE, l'adresse à laquelle le programme est chargé est randomisée
  - Si le noyau ne supporte pas le PIE, le programme est chargé à une adresse par défaut
  - flags de compilation : `-fpie` **et** `-pie`
  - pour vérifié si un programme est compilé avec PIE :

```
readelf -h -d /usr/sbin/smbd | grep 'Type:.*DYN'
Type:      DYN (Shared object file)
Type:      EXEC (Executable file)
```

Exec Shield

- Résumé des fonctionnalités :
  - Protection à l'exécution :
    - mémoire de donnée : flag non-exécutable
      - Si bit NX en HW, utilisation du HW.
      - Sinon, utilisation de l'émulation par "segment limit".
  - Protection à l'écriture
    - mémoire d'instruction : flag non-modifiable
  - ASLR (stack, heap, shared libraries)
  - Position Independent Exécution (PIE)
  - Autres comme
    - glibc security checks (protection contre les format string bugs, manipulations du heap)
    - GCC Fortify source (-D_FORTIFY_SOURCE=2)
    - stack-protector
- Supprime pas mal d'exploits...
- Pas de re-compilation nécessaire
  - mais quelques programmes ne sont pas compatibles... par exemple Xemacs
- Doc officielle : http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf

ExecShield : activation/désactivation

- Sur Linux (privilèges root) :
  - echo 0 > /proc/sys/kernel/exec-shield
  - echo 0 > /proc/sys/kernel/exec-shield-randomization
- Sur Windows, éditer le système de démarrage et remplacer :
  - OptIn with alwaysoff (/noexecute=AlwaysOff).

PaX

- Développement entre 2000 et 2005.
- Caractéristiques
  - PAGEEXEC (émulation du bit NX en utilisant la TLB)
  - SEGMEXEC (émulation du bit NX sur IA-32, segment l'espace mémoire en 2 zones : exec vs non-exec)
  - mprotect() (W^X, zone mémoire soit modifiable, soit exécutable, jamais les deux)
  - Emulation des trampolines GCC
  - ASLR (stack, heap, shared libs,)
  - PIE
- Ne protège pas contre :
  - format string bugs
  - buffer overflows
    - pas de stack guard, pas de stack protector
    - protection contre exécution de la stack...
- Vulnérabilité : escalation de privilège...
- Plus de développement (il semble juste y avoir de la maintenance)

Options GCC : rappel

- -fstack-protector-all
- -fstack-protector-all -Wstack-protector --param ssp-buffer-size=4
- -fpie -pie
- -Wall -Wextra
  Turn on all warnings to help ensure the underlying code is secure.

Options GCC : Non-exec

- Par défaut, gcc compile de manière à ce que la **pile** soit **non-exécutable**
- Exemple :
  - `gcc check_pwd_obo.c -o check_pwd_obo`
  - `gcc -z execstack check_pwd_obo.c -o check_pwd_obo_exec`
  - `readelf -a check_pwd_obo >> check_pwd_obo.readelf`
  - `readelf -a check_pwd_obo_exec >> check_pwd_obo_exec.readelf`
  - `diff check_pwd_obo.readelf check_pwd_obo_exec.readelf`

```
70c70
<   GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
---
>   GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4
```

- L'exécutable peut être modifié à l'aide du binaire « execstack »

GCC : Fortify-source

- Grâce au code source, GCC peut être au courant de la taille d'un buffer. Il pourrait donc éviter des overflows.
- GCC va remplacer strcpy, memcpy, memset par des versions limitées. Par exemple, il remplacera :
  strcpy(dst, src)
  par
  strcpy_chk(dst, src, dstlen)
  et dstlen sera la taille de dst (trouvée par la compilateur).
- En cas de dépassement de tampon :
  - *** buffer overflow detected ***: ./app terminated

Aperçu

Attention à l'ASLR

- Un exécutable compatible avec ASLR, compilé en PIE,
  - "the overhead for PIE on 32bit x86 is up to 26% for some benchmarks."
  - Source : « Too much PIE is bad for performance »,
- Les distributions n'utilise pas le PIE sur tous les binaires.
- Ubuntu indique :
  - "PIE has a large (5-10%) performance penalty on architectures with small numbers of general registers (e.g. x86), so it should only be used for a …"
- Fedora (et RedHat) :
  - Ils maintiennent une liste des packets à sécuriser avec GCC (donc pas tous).

Aperçu

Librairies sécurisées

- Utiliser les fonctions suivantes :
  - utiliser str**n**cpy au lieu de strcpy
  - utiliser s**n**printf au lieu de sprintf
  - utiliser str**n**cat au lieu de strcat
  - utiliser fgets(stdin, str, 10) au lieu de gets(str)
  - utiliser scanf("%10s", str) au lieu de scanf("%s",str)
  - et bien d'autres...
- Libsafe (à présent obsolète)
- ...