Les *buffer overflows*

Définition, Mise en évidence du problème, Solutions

DCSSI – Avril 2005

Plan de la présentation

- Introduction aux vulnérabilités des programmes informatiques
- Description du problème des BO
- Exploitation des BO
- Solutions et prévention des BO

Introduction

- Éléments d’analyse des risques
- Définition de ce qu’est une vulnérabilité informatique
- Typologie et caractéristiques des vulnérabilités

Introduction

- La sécurité vise à limiter des risques à un niveau acceptable
- Un risque est fonction d'une menace, de vulnérabilités et d'éventuelles contre-mesures
- Le risque va influencer sur les paramètres de:
  - Confidentialité
  - Intégrité
  - Disponibilité de l'information

Analyse des risques

- Formule usuelle :

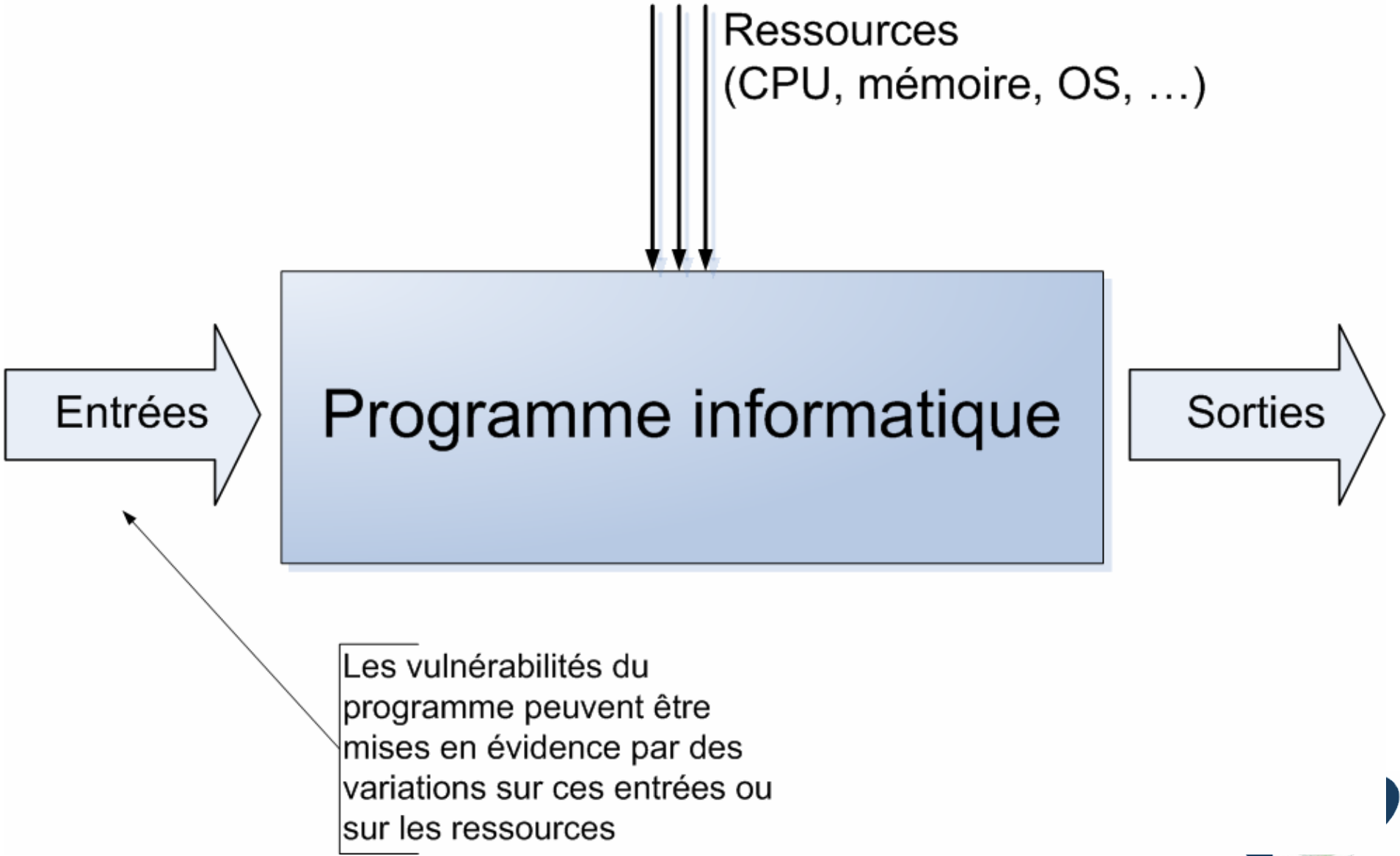
$$Risque = \frac{Menace \times Vulnérabilités}{Contremesures}$$

  - Le potentiel de la menace est augmenté par les vulnérabilités
  - L'impact est réduit par d'éventuelles contre-mesures
  - Permet d'établir *a priori* une hiérarchie des risques

Qu’est ce qu’une vulnérabilité ?

- Une vulnérabilité est une faiblesse inhérente d’un objet
- En informatique, on peut distinguer:
  - Fautes de conception
  - Fautes d’implémentation (implantation)
  - Fautes d’utilisation

Mise en évidence des vulnérabilités



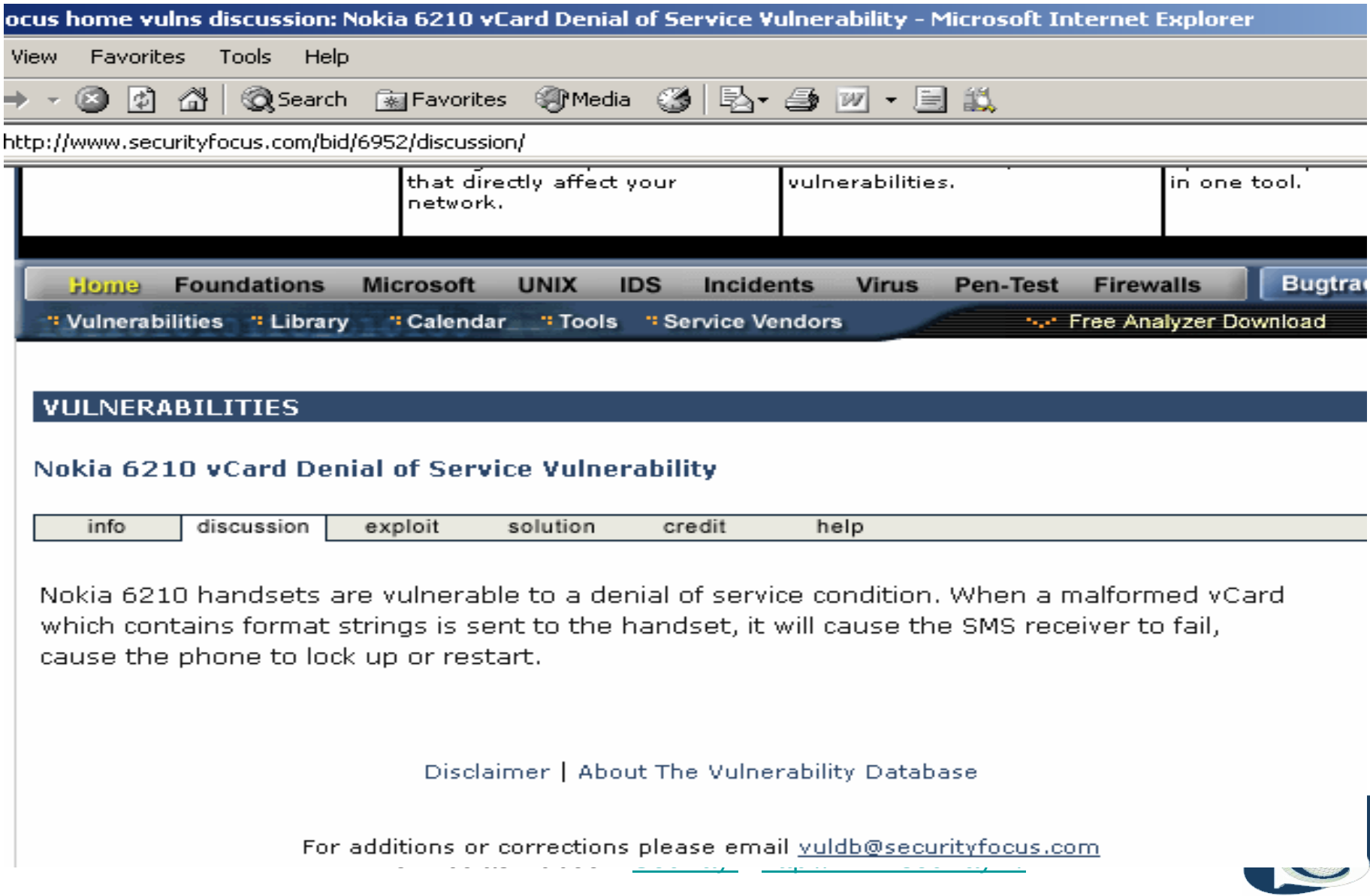
Fautes de conception

- Système vulnérable avant même sa réalisation technique
- Exemples de vulnérabilités inhérentes des algorithmes de chiffrement
  - Involontaires : algorithmes faibles par conception
  - Volontaires : introduction de portes dérobées dans l'algorithme
- Exemple de vulnérabilité d'un système informatique
  - Les imprimantes réseaux sont souvent configurables par Web, certains modèles ne peuvent pas être protégés par mot de passe
  - La conception n'a pas pensé qu'un mot de passe serait nécessaire pour protéger l'interface Web
  - Possibilité de reconfigurer l'imprimante à distance

Fautes d'implémentation (1)

- Aussi appelées Fautes d'implantation ou de développement
- Vulnérabilités introduites (volontairement ou non) dans le code source du système
- Lors d'une introduction volontaire de vulnérabilités, on parle de *backdoor*
- Lors d'une introduction involontaire, on parle de *faille*

Fautes d'implémentations (2)



ocus home vulns discussion: Nokia 6210 vCard Denial of Service Vulnerability - Microsoft Internet Explorer

View Favorites Tools Help

Search Favorites Media

http://www.securityfocus.com/bid/6952/discussion/

that directly affect your network. | vulnerabilities. | in one tool.

Home Foundations Microsoft UNIX IDS Incidents Virus Pen-Test Firewalls Bugtra

Vulnerabilities Library Calendar Tools Service Vendors Free Analyzer Download

VULNERABILITIES

Nokia 6210 vCard Denial of Service Vulnerability

info | discussion | exploit | solution | credit | help

Nokia 6210 handsets are vulnerable to a denial of service condition. When a malformed vCard which contains format strings is sent to the handset, it will cause the SMS receiver to fail, cause the phone to lock up or restart.

Disclaimer | About The Vulnerability Database

For additions or corrections please email vuldb@securityfocus.com

Conséquences des failles

- Selon leur gravité, ces problèmes peuvent être utilisés pour:
  - Récupérer de l'information
  - Faire planter le système affecté
  - Prendre complètement le contrôle du système affecté
- Dans ce dernier cas, on dit que la vulnérabilité est « exploitable »

Remote et *Local exploits*

- Dans le jargon sécurité, un programme d'attaque utilisant une faille d'un système pour en prendre le contrôle ou le faire planter est appelé un « *exploit* ».
- On parle de:
  - *Remote exploit* lorsque l'attaque est possible à distance
  - *Local exploit* lorsqu'il faut un accès préalable au système avant de pouvoir lancer l'attaque

Comment sont découvertes les vulnérabilités ?

- Recherche active de failles par la communauté sécurité
  - Étude du code source ou des binaires
  - Équipes d'audit de sociétés informatiques, spécialistes sécurité ou éditeurs de logiciels
  - Groupes de pirates
- Lorsque faille rendue publique, l'éditeur du logiciel affecté sort un « patch » (correctif)
- Les pirates essaient souvent de garder les failles privées : *0day exploit*.

Grandes familles de failles de programmation

- Non filtrage des métacharactères dont:
  - Injection SQL
  - Scripts CGI/Perl avec appels Shell
- *Format strings*
- *Buffers overflows*

Les buffers overflows

- **Introduction aux BO**
- Rappels sur la mémoire et les microprocesseurs
- Exploitation des BO
- Solutions

Introduction

- *Buffer* = zone mémoire réservée par un programme pour y stocker/lire/écrire des données
- Les buffers overflows (BO) sont une grande famille de failles d'implémentation
- La plupart des failles publiées depuis des années sont des BO ou assimilés
- Premier BO connu : 1989, le Morris Worm utilise un BO dans finger sous Unix
- Redécouvert en 1995 par Thomas Lopatic (DE) avec démonstration (exploit) sur HP-UX

Petite anecdote…

- La première exploitation connue à grande échelle d'un BO a été le "Morris Worm"
- Ver (virus) programmé par le fils d'un directeur de la recherche de la NSA
- …

Aperçu général des BO

- Un buffer overflow est une faute d'implémentation consistant à déborder de la mémoire allouée pour une opération
- La plupart des BO dus au programmeur ne vérifiant pas les tailles mémoires avant d'effectuer des opérations de copie
- Les BO sont très courants dans les programmes écrits en langages dits « bas niveaux » comme le C ou C++

Conditions d'arrivée d'un BO

- Plateforme et langages de programmation ne vérifiant pas les opérations mémoire faites par le programmeur
  - C, C++ pour des raisons de rapidité
  - C'est au programmeur de s'assurer de la validité des opérations qu'il fait !
- Opérations mémoires de lecture/écriture/copie
- Non (ou mauvaise) vérification des opérations mémoires

Exemples de fonctions C pouvant générer des BO ...

- …si on ne vérifie pas avant de les appeler
- strcpy()
- strcat()
- sprintf()
- gets()
- scanf(), fscanf(), sscanf()
- Potentiellement, toute fonction écrite par le développeur et faisant des accès à la mémoire

Exemple de BO (1)

- Programme en langage C :

```
char *nom;
char *buf;
…
buf = (char *) malloc(1024);
scanf('%s', buf);
…
```

Crée une variable « buf »

Réserve 1024 octets de mémoire pour « buf »

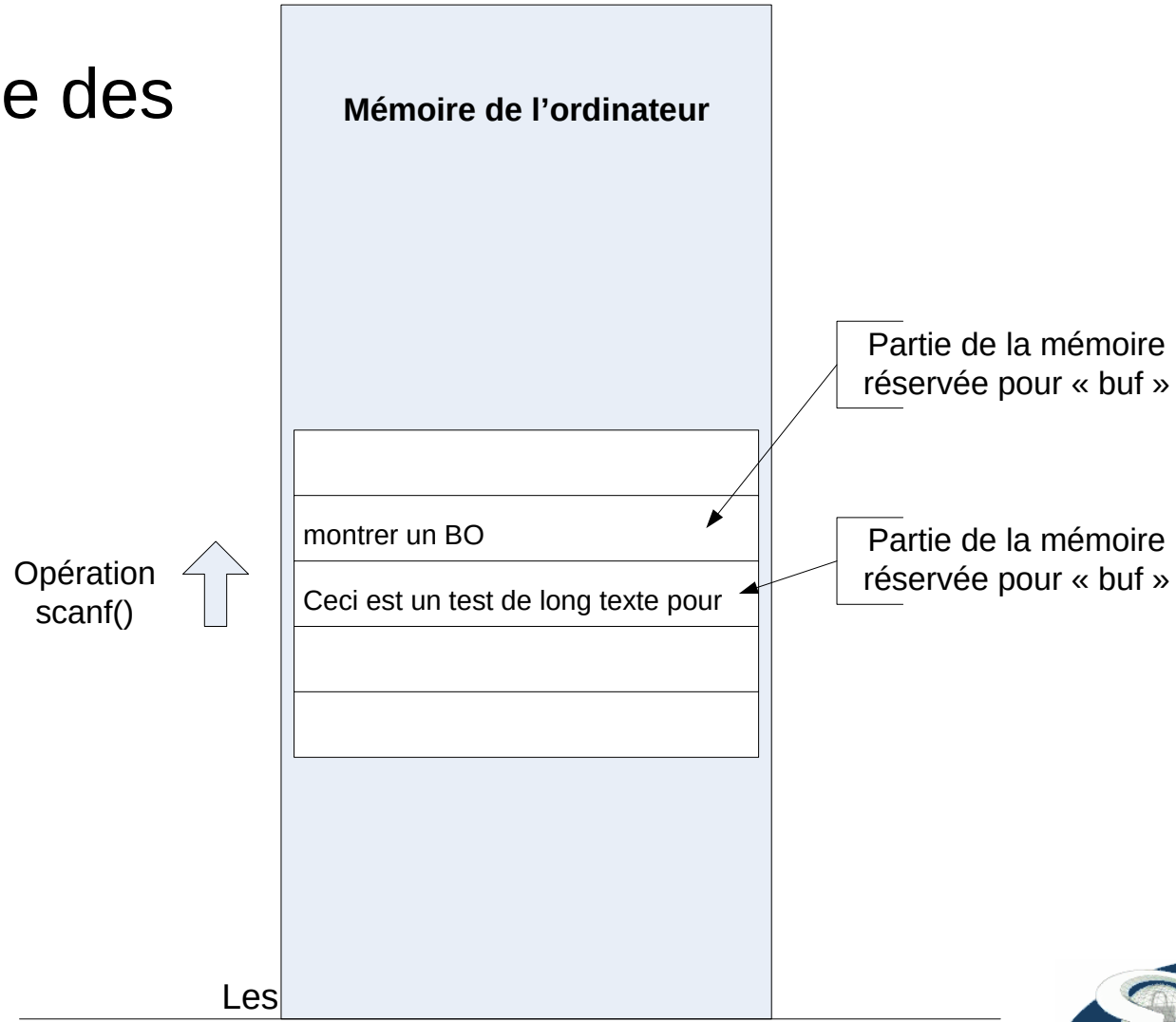
Demande à l'utilisateur de rentrer du texte au clavier, met ce texte dans buf.

Exemple de BO (2)

- Constats sur le programme précédent
  - On alloue 1024 octets
  - On demande à l'utilisateur de rentrer du texte et on le met dans les 1024 octets réservés précédemment
- La fonction utilisée « scanf » ne vérifie pas si la taille du texte est bien inférieure à 1024 octets
- Dans le cas contraire, scanf continuera à copier le texte tapé en mémoire après nos 1024 octets !

Exemple de BO (3)

- Vue mémoire des opérations



Conséquences d'un BO

- Des informations en mémoire sont potentiellement écrasées
- Au mieux, crash du programme
  - Arrivée à la fin de la mémoire réservée au programme
  - Écrasement de zones vitales pour le programme
- Au pire, manipulation de cases mémoires importantes et incidences sécurité

Exploitation des BO

- L’exploitation revient à écraser des zones mémoires sensibles
  - Adresses de retour ou pointeurs sur fonctions
  - Variables mémoires importantes (nom d’utilisateur, …)
  - Variables internes du système ou programme

Les buffers overflows

- Introduction aux BO
- **Rappels sur la mémoire et les microprocesseurs**
- Exploitation des BO
- Solutions

Rappels gestion de la mémoire (1)

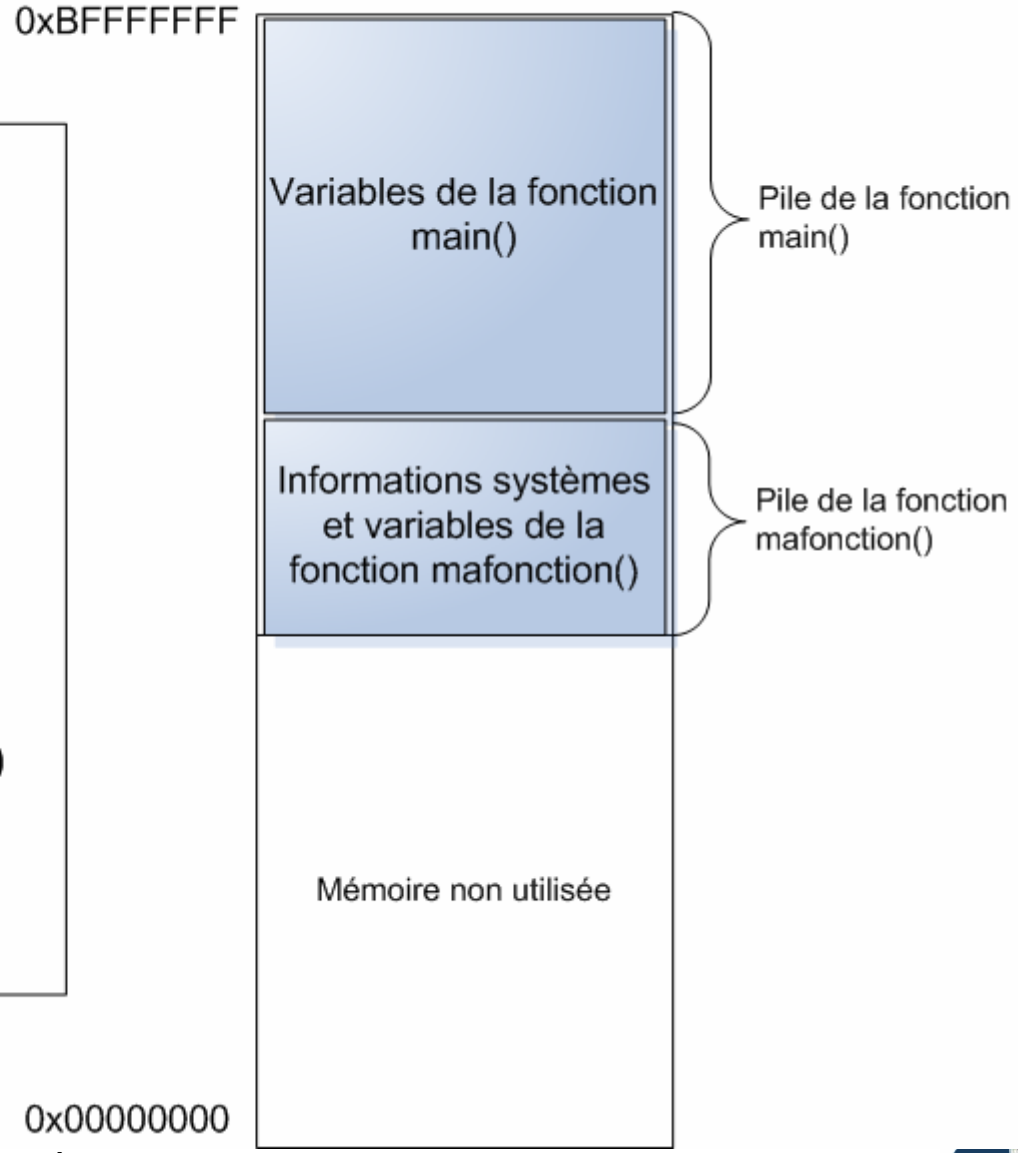
- Un programme a des besoins en mémoire qu'il peut adresser de plusieurs façons différentes
- Les architectures classiques utilisent deux types de mémoire
  - La pile (*stack*) pour les variables dites automatiques
  - Le tas (*heap*) pour les espaces mémoires globaux alloués dynamiquement
  - Le programmeur dispose de la pile et du tas comme il le souhaite
- Les buffers overflows écrasent donc des parties de la pile ou du tas, selon le code du programme

La pile (stack) (1)

- La pile est un emplacement mémoire où l'on peut mettre (empiler), lire, modifier, enlever (dépiler) des éléments
  - Les piles descendent dans la mémoire
  - Technique LIFO: Last In, First Out
  - Opérations PUSH (empiler un élément) et POP (dépiler un élément)
- Chaque fonction a, au moment de son exécution, son propre morceau de pile (stack frame)
  - A la fin de la fonction, destruction de sa pile
- Les compilateurs s'en servent pour mettre les variables dynamiques des fonctions
- Sur la pile sont aussi stockées des informations systèmes (cf. plus loin)

Pile d'une fonction

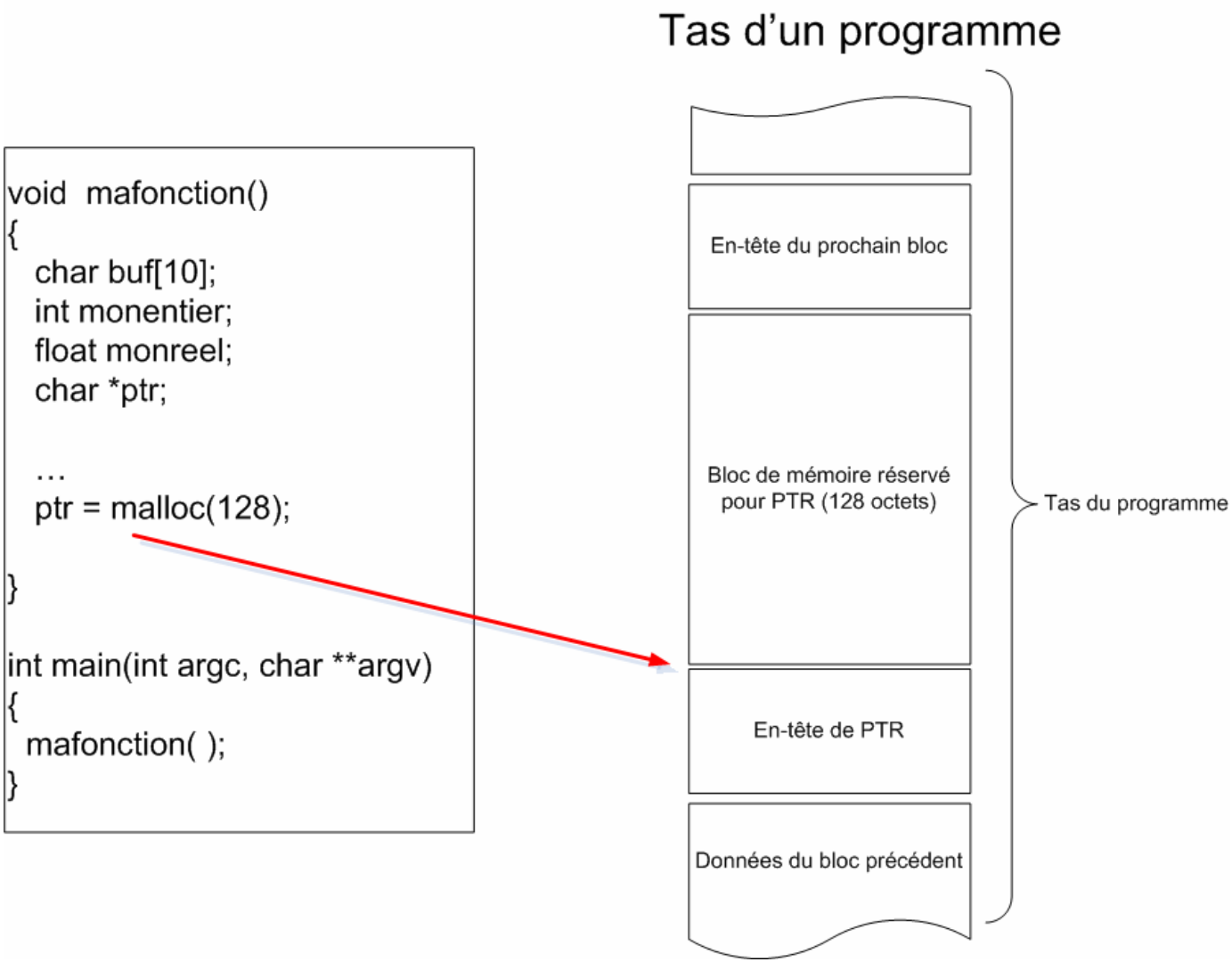
```
void  mafonction()
{
   char buf[10];
   int monentier;
   float monreel;

   …

}

int main(int argc, char **argv)
{
  mafonction( );
}
```



Le tas (heap) (1)

- Le tas est une autre partie de la mémoire globale du programme
- Le plus souvent situé après la fin du programme et de ses données
  - Organisé sous forme de « listes »
- Les fonctions peuvent réserver des parties du heap en appelant malloc
  - Mémoire allouée préservée à la sortie de la fonction
- (*ptr=malloc(taille)*) et libérer cette mémoire avec free (*free(ptr)*)

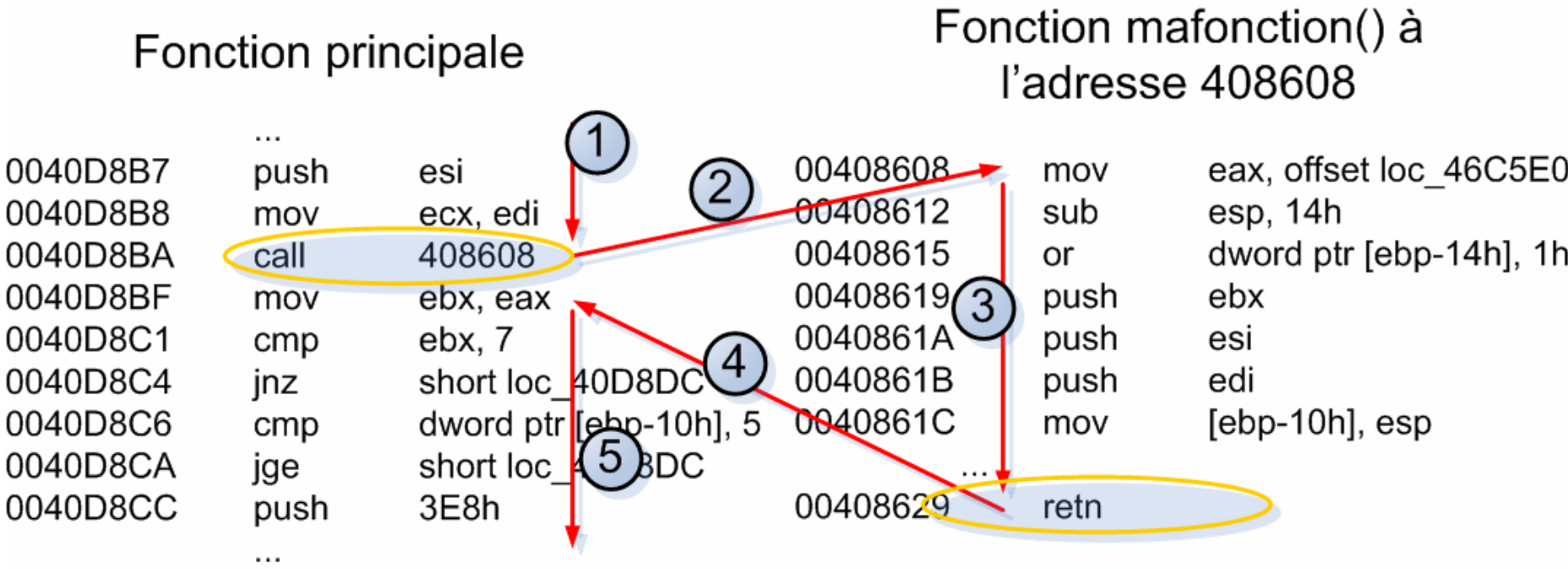
Le tas (heap) (2)



Rappels sur les fonctions

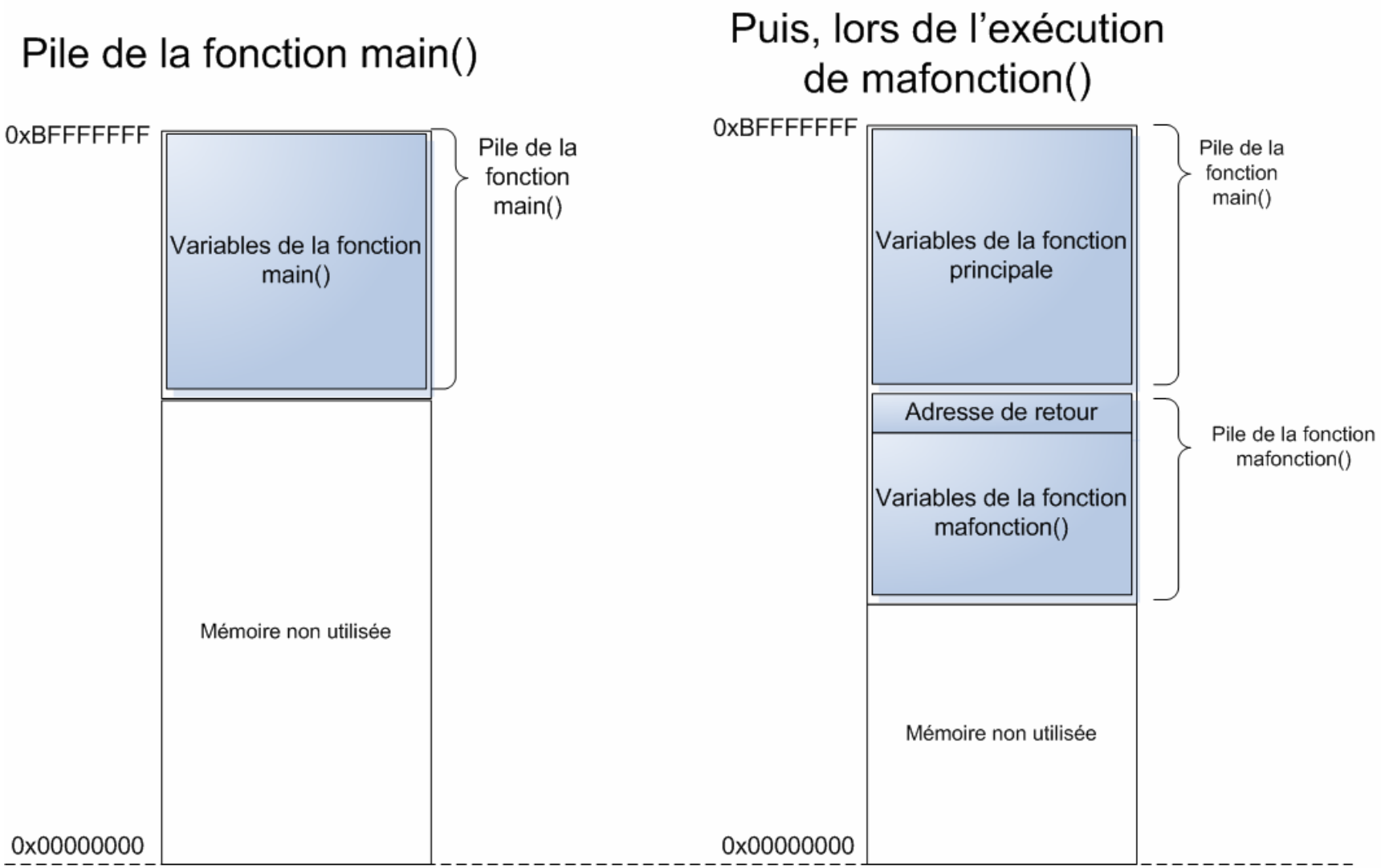
- On regarde maintenant au niveau CPU
  - Langage C -> Assembleur -> CPU
- Appel d'une fonction
  - le CPU empile l'adresse de retour
  - Instruction CALL, fait implicitement un PUSH
- Retour d'une fonction (return en C)
  - le CPU dépile l'adresse, positionne le pointeur d'instruction courante à cette adresse
  - Instruction RET, fait implicitement un POP
- Adresse de retour = adresse juste après le CALL dans la fonction qui appelle

Illustration niveau CPU



1 = Flux d'exécution de la fonction principale avant l'appel à 408608
2 = « CALL », Appel de la fonction située à 408608, on « empile » l'adresse de retour (c.-à-d. 40D8BF) et on va à l'adresse 408608.
3 = Exécution de la fonction 408608
4 = « RET », Fin de la fonction située à 408608, on « dépile » l'adresse de retour et on va à cette adresse
5 = La fonction principale continue

Illustration niveau mémoire



Les buffers overflows

- Introduction aux BO
- Rappels sur la mémoire et les microprocesseurs
- **Exploitation des BO**
- Solutions

Techniques d’exploitation

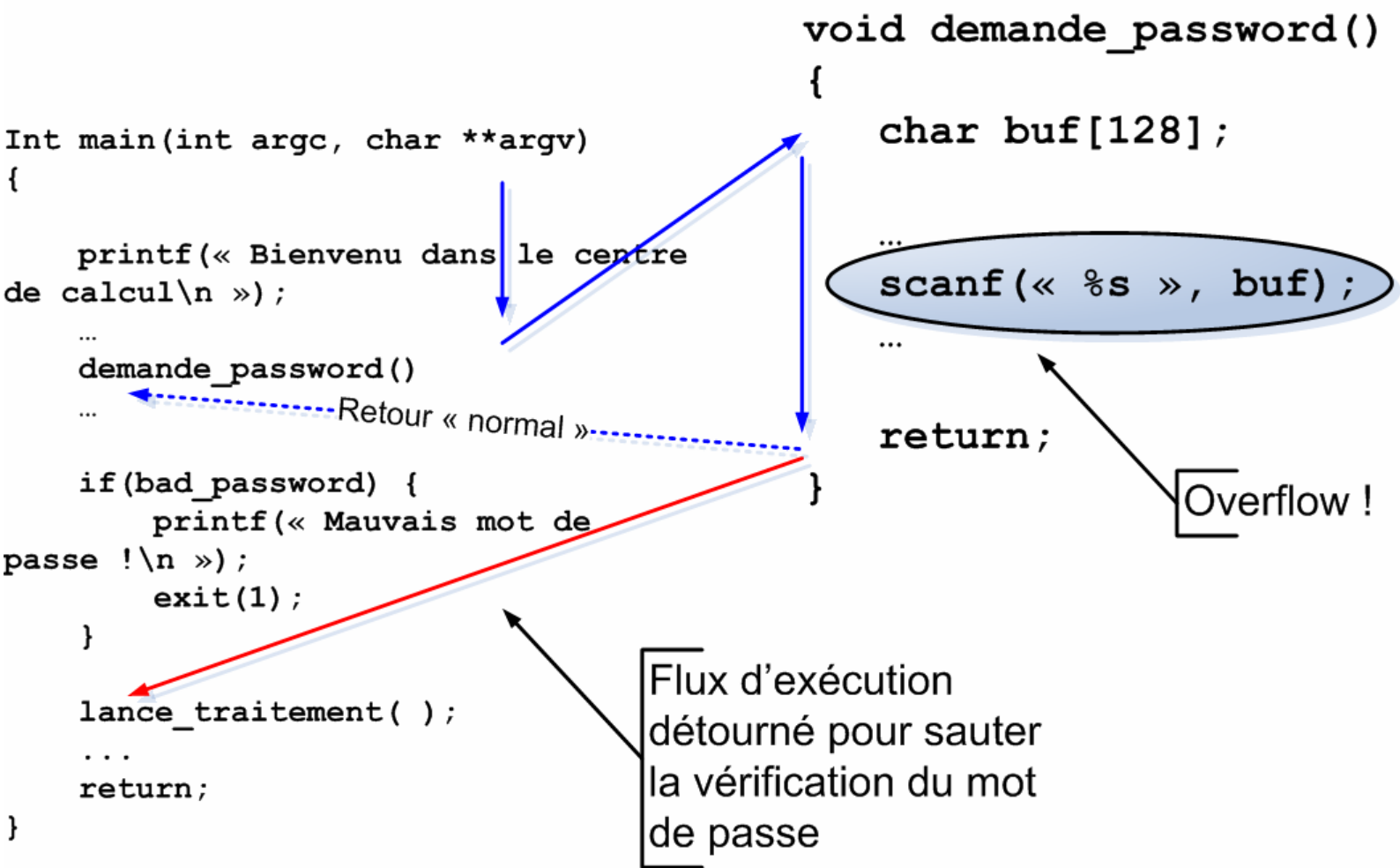
- Buffers dans la stack
  - Modification de variables importantes sur la stack
  - Modification de l’adresse de retour vers autre partie du programme
  - Modification de l’AR vers un buffer en stack
  - Modification de l’AR vers un buffer hors stack
  - Modification de l’AR vers fonction de la LIBC
- Buffers dans le heap

Les stack overflows (1)

- Le buffer vulnérable est situé sur la stack d'une fonction
- Le pirate peut faire déborder ce buffer avec des données qu'il contrôle
- Il peut donc aller écraser l'adresse de retour de la fonction
- Et ainsi, détourner le programme lors du RET
  - Retour non pas vers la fonction appelante, mais vers n'importe quel endroit de la mémoire !

Les stack overflows (2)

- Retour vers le programme lui-même



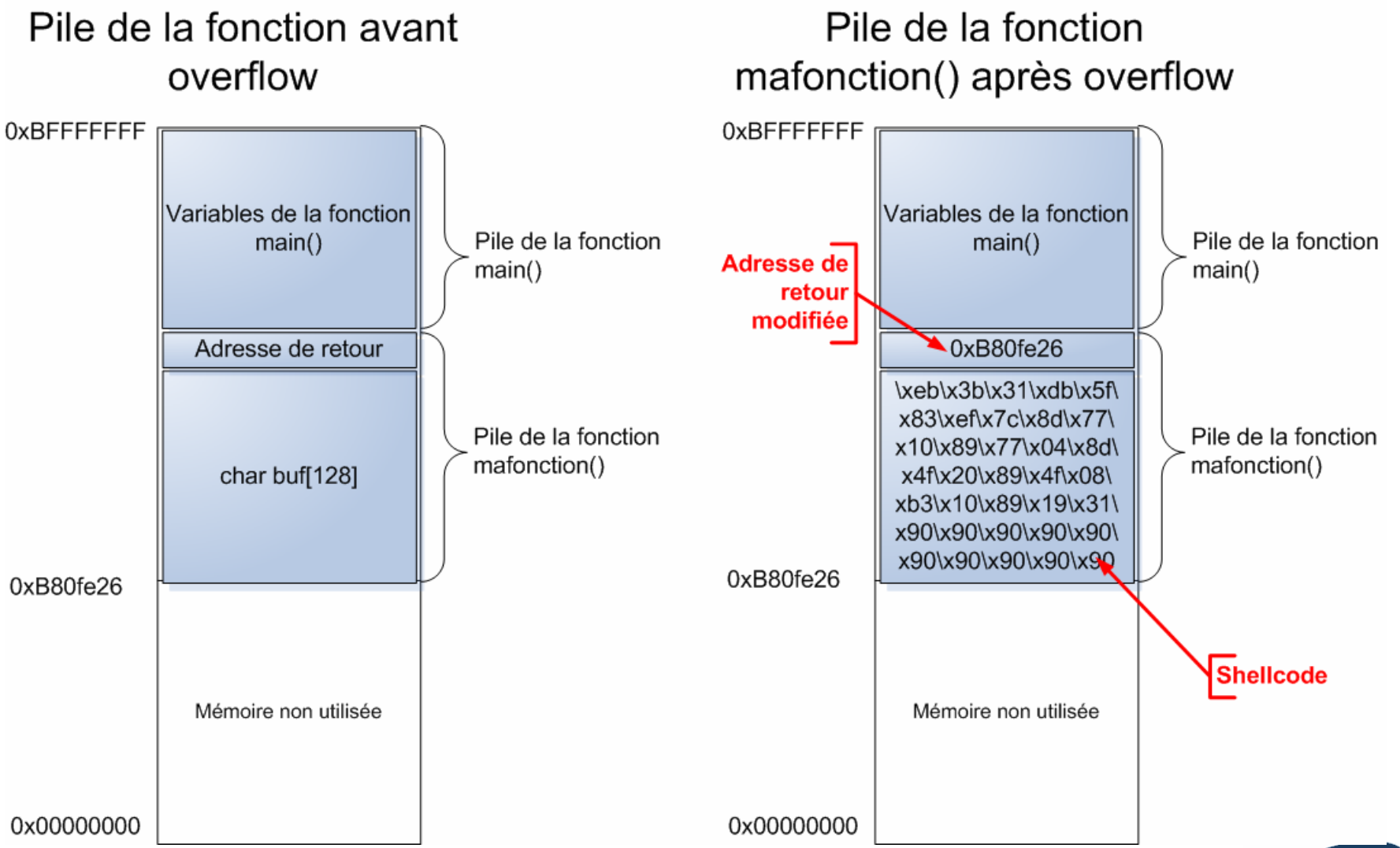
Les stack overflows (3)

- Retour du programme vers une partie de la mémoire que l'on contrôle
- Par exemple, le début du buffer vulnérable
- On peut y placer notre propre code assembleur, souvent appelé *shellcode*
- Qui sera exécuté au RET de fin de fonction !

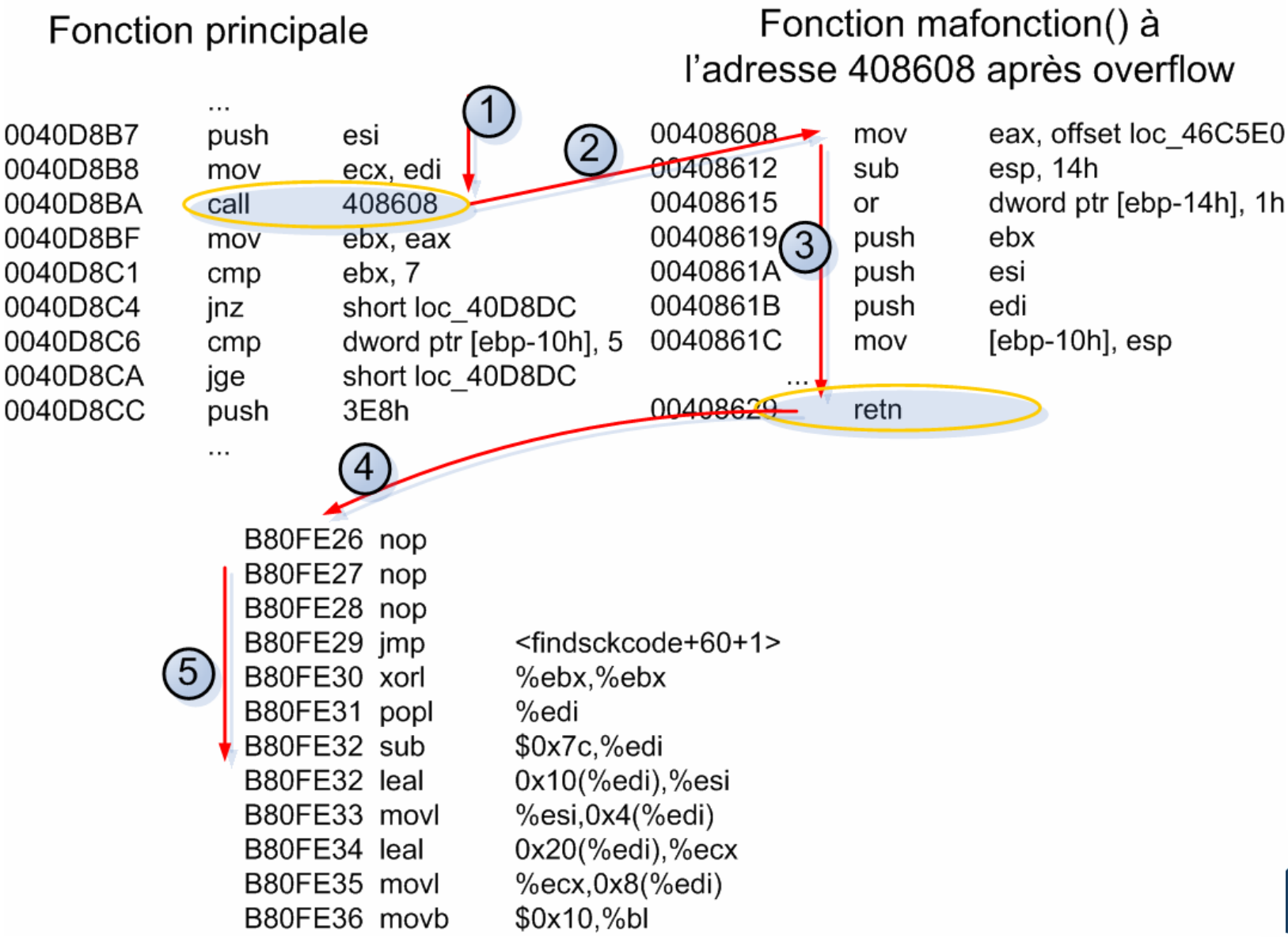
Le shellcode

- Code machine (binaire), souvent développé en assembleur
- Va être exécuté par le CPU à la suite de l'overflow
- Très souvent, ce code exécute un shell Unix
- Afin de palier au problème de la prédiction des adresses, de nombreuses instructions NOP (0x90) mises au début de ce buffer

Les stack overflows (4)



Les stack overflows (5)



Les stacks overflows (6)

- Une protection courante contre ce type d'exploit consiste à interdire l'exécution de code sur la stack
- Protection contournée par des variations sur l'adresse de retour:
  - Au lieu de la stack, on retourne vers un autre endroit que l'on contrôle
  - On peut aussi retourner directement dans des fonctions du système d'exploitation (*Return into LibC*)

Les stacks overflows (6)

- Return into LibC: Exemple de Linux
  - Les fonctions systèmes sont appelées au travers d'une librairie dite *libc*
  - Les arguments des fonctions appelées sont mis sur la stack
  - On peut donc overflower notre buffer, mettre l'adresse d'une fonction libc à la place de l'adresse de retour, puis les arguments
- Protection: « randomization » des adresses libc

Les heap overflows (1)

- Les gestionnaires de heap mettent des informations devant/derrière chaque bloc
- Possibilité d'écraser ces informations
- Selon comment le gestionnaire utilise ces informations, possibilité d'écraser de la mémoire
- Le + souvent, attaque indirecte sur l'adresse de retour en stack:
  - On met dans ces en-têtes l'adresse dans la mémoire de l'adresse de retour sur la stack (ouf!)
  - La prochaine opération type malloc/free va modifier l'adresse de retour selon nos desiderata

Les heap overflows (2)

- Fréquents sous Windows car OS et nombreuses applications écrites en C++
- Les objets créés avec new en C++ sont placés sur le tas
- L'exploitation des HO est aisée sous Windows, car le tas contient beaucoup d'informations importantes
  - SEH
  - Pointeurs de fonctions…

Les buffers overflows

- Introduction aux BO
- Rappels sur la mémoire et les microprocesseurs
- Exploitation des BO
- **Solutions**

Prévention vs. protection

- Les seules méthodes complètement sûres seraient de:
  - De bien programmer
  - Les rendre impossibles de façon inhérente (langages avec vérification)
- Méthodes réalistes:
  - Auditer le code de manière proactive
  - Rendre l'exploitation extrêmement difficile

Écriture sécurisée

- La complexité des développements est déjà assez conséquente
- Les programmeurs ne disposent souvent pas des compétences sécuritaires
- L'écriture sécurisée relève d'un véritable art
  - Même les produits de sécurité sont affectés, ex: ISS RealSecure, récentes failles Checkpoint FW-1
- Les possibilités d'introduire involontairement des failles sont nombreuses
- La sémantique même des langages est parfois trompeuse
  - Exemples : strncpy(dst, src, nbr), snprintf(…)

Audit proactif de code (1)

- Logiciels ayant du vécu
  - Nombreux audits, chacun ayant apporté leurs découvertes
  - Failles encore découvertes mais plus rarement et surtout très complexes (ex: Sendmail < 8.12.8, < 8.12.9, < 8.12.10 17/9/2003)
  - Cas particulier Windows: failles non complexes mais protocoles et programmes complexes
- Nouveaux logiciels
  - De nombreux logiciels intègrent maintenant dès leur origine la programmation sécurisée
  - Mais les erreurs se répètent… ex: OpenSSH chall < 3.4

Audit proactif de code (2)

- Le projet OpenBSD
  - Développer un système d'exploitation orienté sécurité
  - Logiciels inclus dans OpenBSD vérifié par une équipe d'auditeurs
  - Très bons résultats, beaucoup moins de failles découvertes sur OpenBSD
  - Système recommandé pour les serveurs Web, DNS, FTP, Mail
  - Quelques failles découvertes par an, problème des logiciels additionnels
- Sardonix
  - Projet pour l'audit du code open source
  - Financé par le DARPA américain
  - Principalement orienté Linux (dérivé du projet LSAP)
  - Mécanisme de classement des auditeurs
  - Échec complet

Audit de code (3)

- Initiative Microsoft
  - Bill Gates: « Security is our biggest challenge ever »
  - Audit proactif du code Windows par équipes internes
  - Aussi auditeurs de sociétés spécialisées, mais travaux tenus secrets (ex: AtStake)
  - Correctifs inclus dans les Service Packs et mises à jour WindowsUpdate, non documentés
  - Intégration par défaut de la sécurité dans les nouveaux OS (Windows 2003 Server, XP SP2, Microsoft Visual Studio .NET)

Méthodes pour l'audit de code source

- Audit de code manuel : inspection des lignes de code par un auditeur
  - Dépend grandement de la compétence de l'auditeur et du temps passé dessus
  - Ex: ISS X-Force, faille Sendmail sept. 2003. Plusieurs jours passés sur quelques dizaines de lignes de code
- Inspection automatisée
  - Logiciels d'analyse du code source
  - Ex: RATS, LINT
  - Souvent peu efficaces, seules failles les plus évidentes découvertes
  - Les outils existants sont peu évolués
  - D'autres méthodes automatisées plus intéressantes
  - Problème de l'audit de logiciels propriétaires

Méthodes pour l'audit de binaires (1)

- Analyse inversée
  - Retrouver le code source à partir du binaire
  - Revient à de l'étude de code source
  - Le langage est alors l'assembleur…
  - Outils désassembleurs (analyse hors ligne) : IDA, WDASM
  - Outils debuggers (analyse en ligne) : SoftIce, OllyDbg, IDA (dernières versions)
  - Exemple de faille découverte par analyse inversée : DCOM/RPC par groupe polonais LSD, juin 2003

Méthodes pour l'audit de binaires (2)

- Outils en ligne
  - *Profilers* permettant de suivre les opérations mémoire
    - Instrument le code et vérifient toutes les opérations mémoire
    - Normalement utilisés pour trouver les « fuites » de mémoire
    - Exemple: Rationale Purify (IBM)
  - *Fuzzers* permettant d'injecter automatiquement des entrées non usuelles
    - Résultats assez intéressants
    - Permettent de découvrir des failles complexes
    - Écriture d'un protocole de test ensuite appliqué à toutes les implémentations d'un même standard ou envoi direct
    - Nécessitent cependant la compréhension complète du protocole d'entrée pour être performants
    - Exemples: Spike (Immunity), PROTOS (University d'Oulu)

Constats

- Les méthodes précédentes sont
  - Coûteuses
  - Difficiles
  - Peu efficaces
- Les failles resteront, il faut essayer de limiter leur impact
- Nous allons maintenant étudier les méthodes permettant de limiter l'impact des failles

Séparation des privilèges (1)

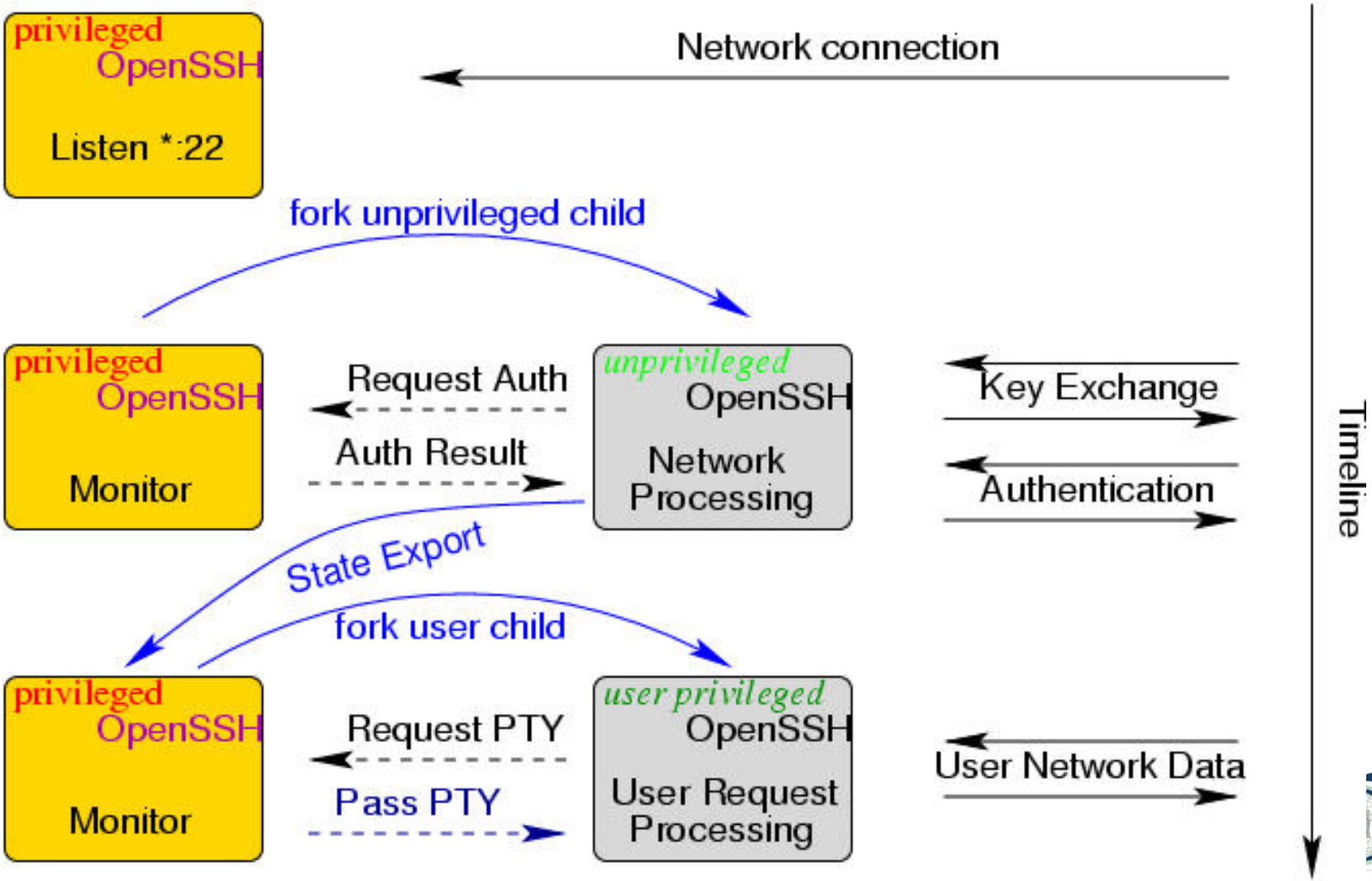
- Un programme réseau peut être découpé en deux parties:
  - Un processus père de contrôle, privilégié (root)
  - Un processus fils de communication réseau recevant et traitant les données, non privilégié et restreint
  - Les deux communiquent par tubes
- Le processus père est très simple, d'où risques d'overflows limités
- Exemple: séparation des privilèges dans OpenSSH, overflows récents

Séparation des privilèges (2)

- N’empêche pas les overflows, mais limite leur portée
- Nécessite la réécriture des logiciels
- Certaines parties du programme seront cependant toujours exposées

Séparation des privilèges (3)

Cas OpenSSH :



Systèmes des canaris – StackGuard (1)

- Modification du compilateur GCC pour intégrer un mécanisme anti-overflow dans chaque appel de fonction
- Disponible sous Linux
- A chaque appel de fonction, un nombre aléatoire (*canari*) est placé sur la stack juste en dessous de l'adresse de retour
- Lors du retour de la fonction, un mécanisme vérifie automatiquement si ce canari n'a pas changé
- En cas de changement, le programme est tué et une alerte est inscrite dans les journaux de l'OS
- Extension : l'AR est elle-même XORée avec le canari

Systèmes des canaris - StackGuard (2)

- Limitations:
  - Impact sur les performances (faibles cependant)
  - Nécessite de recompiler les programmes
  - Ne protège que l'AR dans les stack overflows
    - En cas d'overflow n'allant pas jusqu'à l'AR, protection inefficace
  - N'agit qu'au retour de la fonction
    - en cas d'overflow qui impacterait le programme avant le retour de la fonction, protection inefficace

Systèmes des canaris - Propolice

- Projet d'IBM Japon utilisant une technique similaire à StackGuard mais étendue
- Utilisé dans les nouvelles versions de l'OS OpenBSD
- En plus des canaris, ProPolice réordonne les variables dans la stack
- En plaçant les buffers après les autres variables (notamment après pointeurs)
- Pour éviter à l'overflow d'écraser des pointeurs

Déplacement de l'AR - StackShield (1)

- Idée initiale : séparation des informations de contrôle (AR) et des données
- Modification du compilateur consistant à enlever l'adresse de retour de la stack
- Pour la stocker dans une partie dédiée de la mémoire
- A chaque entrée dans une fonction, copie de l'AR hors de la stack; à chaque sortie, recopie vers la stack

Déplacement de l'AR – StackShield (2)

- Limitations:
  - Ne protège pas contre la corruption de données sur la stack
    - Dont potentiellement des pointeurs pouvant être utilisés pour écraser d'autres données en mémoire, comme fnlist de on_exit()
  - Ne protège pas contre les heap overflows
    - De plus en plus courants depuis papiers expliquant comment les exploiter
  - StackShield n'est plus maintenu, donc pas à utiliser

Stack non exécutable (1)

- Méthode introduite par Solar Designer (http://www.openwall.com/)
- Consiste à interdire l'exécution de code sur la pile
- L'endroit de la mémoire où est la pile est marqué comme « non exécutable »
- En cas d'exécution, une exception est déclenchée, le programme est tué, une alerte est enregistrée
- Implémenté au niveau du processeur et des systèmes d'exploitation
  - Linux (patch OpenWall pour les noyaux Linux)
  - Solaris (activé dans /etc/system)
  - Autres systèmes Unix propriétaires

Stack non exécutable (2)

- Limitations:
  - Ne protège pas contre les heap overflows
  - Ne protège pas contre l'exécution de code hors de la stack (dans le tas par ex.)
  - Nécessite un noyau modifié
  - Incompatible avec certains compilateurs (trampolines GCC) et gestion des signaux Linux, le patch prend cependant en compte ces cas

Tas non exécutable

- On empêche toute exécution de code dans le tas
- Limitations:
  - Ne prévient pas les retours dans la libc
  - Ne prévient pas contre les retours dans la stack
  - Nécessite un noyau modifié

Randomization des segments

- Afin de limiter l'exécution de code hors stack, possibilité de placer les segments aléatoirement en mémoire
- Lors du démarrage du programme, l'OS choisit aléatoirement où placer les différentes sections de programme
- Ainsi, le pirate ne peut prévoir où se situera le code qu'il veut exécuter

En pratique…

- Prises indépendamment, les solutions précédentes sont perfectibles
- On utilise donc une combinaison de ces techniques
- Des implémentations de ces techniques sont fournies sous forme de patchs pour GCC ou de patchs pour les kernels

Solutions actuelles (1)

- Patchs OpenWall pour Linux
  - http://www.openwall.com/linux/
  - Stack non exécutable + d'autres modifications
- StackGuard
  - http://www.immunix.org/stackguard.html
  - Modification de GCC pour introduire des canaris
- Système d'exploitation OpenBSD 3.4
  - http://www.openbsd.org/
  - SSP (nouveau nom de ProPolice), canaris
  - W^X, mémoire soit modifiable, soit exécutable
- Patchs PaX pour Linux
  - http://pax.grsecurity.net/
  - Implémentation disponible pour Linux sur processeurs x86
  - Mémoire soit modifiable, soit exécutable
  - Randomization des mappings mémoire

Solutions actuelles (2)

- Certains Linux intègrent directement les modifications OpenWall, StackGuard ou PaX:
  - OpenWall: Owl Linux (http://www.openwall.com)
  - StackGuard: Immunix (http://www.immunix.org/)
  - PaX: GRSEC (http://www.grsecurity.net/)
- Cas particulier de Windows:
  - Les nouveaux compilateurs Visual Studio .NET offrent la possibilité d'un mécanisme de canaris
  - Tout le code de Windows 2003 Server est compilé avec ces canaris

Conclusion

- Les buffers overflows, un problème réel
- Des techniques d'attaque d'une sophistication extrême
- Auxquelles répondent des techniques de protection complexes !
- Pour couvrir un réel besoin, en attendant des programmes et systèmes sans bugs…