

DE LA RECHERCHE À L'INDUSTRIE



Sécurité des Systèmes Unix

Sécurisation matérielle, système et logicielle

Commissariat à l'Énergie Atomique et aux Énergies Alternatives
Romain CARRÉ

7 novembre 2019

- 1** Préambule
 - Sécurité physique
- 2** Sécurité système
 - Authentification
 - Autorisation
 - Audit
- 3** Sécurité applicative
 - Menaces
 - Contre-mesures
- 4** Installation d'un bastion

- 1** Préambule
 - Sécurité physique
- 2** Sécurité système
 - Authentification
 - Autorisation
 - Audit
- 3** Sécurité applicative
 - Menaces
 - Contre-mesures
- 4** Installation d'un bastion

- Concerne l'environnement du système et les étapes préparant son démarrage, notamment :
 - piégeage des composants
 - séquence de boot
 - rayonnement électromagnétique
- Aspects périphériques non techniques :
 - corbeilles, tableaux blancs, imprimantes partagées
 - risques d'incendie, électriques, inondations
- Objectif :
 - meilleure évaluation de la cohérence de la sécurité

- Pourquoi placer les mécanismes de sécurité dans les couches inférieures du système ?
 - augmenter l'assurance grâce à la simplicité des mécanismes
 - réduire la dégradation de performance grâce à l'intégration
- Une vulnérabilité dans les couches inférieures peut annuler les efforts de sécurisation des couches supérieures (effet « court-circuit ») :
 - vol de disques
 - attaque sur les couches réseaux (ARP par exemple)
 - attaque système qui touche plusieurs applications
(bibliothèque standard par exemple)



- Exemples d'attaques par les composants matériels :
 - Attaques sur le bus Firewire (2005)
 - R/W sur la mémoire vive
 - déverrouillage de session, passage de processus en UID 0
 - Attaques sur la SMRAM (system management)
 - emplacement idéal pour un rootkit discret
 - détection quasi-impossible
 - Attaques sur le bus PCI
 - espionnage d'autres composants, injection de données
 - R/W sur la RAM (encore ! tellement efficace...)
 - Nouvelles technologies pour l'administration des machines
 - Intel Active Management Technology, Computrace, ...
 - possibilité d'installer un système entièrement à distance (KS)

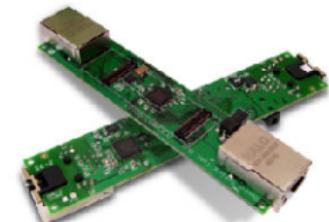
■ Sécurité du boot

- BIOS : contrôle des périphériques de démarrage
- OS Loader : verrouiller la configuration du système, sinon
 - linux init=/bin/bash
 - Windows (F8) : mode debug
- Chiffrement de disque
 - mais... copie de la RAM à chaud
 - mais... piégeage du loader avec un keylogger
- Puces TPM (Trusted Platform Module), UEFI SecureBoot
 - possibilité de stocker une empreinte de la séquence de démarrage, du BIOS au noyau
 - mais... complexité de la mise à jour du système
 - mais... attaques matérielles possibles

- Méthodes à 2€ (environ)
 - Keylogger hardware



- Module de capture sur VGA, DVI ou HDMI
- PC miniature intégré et caché
- Attaques dites « de la femme de ménage »

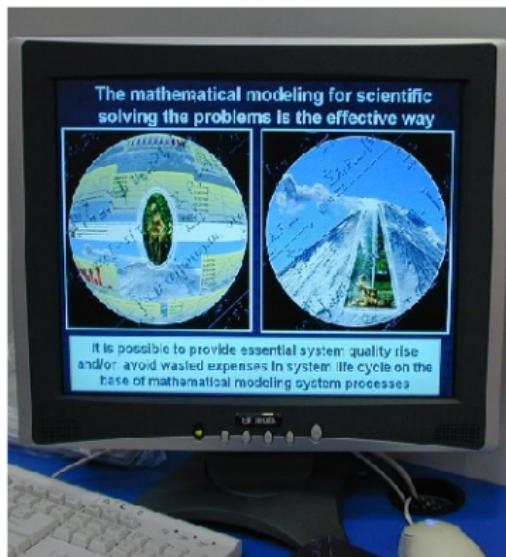


■ Rayonnements compromettants

■ Effet « TEMPEST »

Tout matériel ou système qui traite ou transmet, sous forme électrique, des informations est sensible à des perturbations électromagnétiques temporaires. Ces perturbations, qualifiées de signaux parasites, sont provoquées par les variations du régime électrique, dans les différents circuits du matériel considéré durant son fonctionnement.

- Ondes électromagnétiques
- Courant de conduction le long de canalisations ou de câbles
- Capture à distance de frappes clavier, d'image sur un écran
- Contre-mesures : cage de Faraday, brouilleurs



- 1** Préambule
 - Sécurité physique
- 2** Sécurité système
 - **Authentification**
 - **Autorisation**
 - **Audit**
- 3** Sécurité applicative
 - Menaces
 - Contre-mesures
- 4** Installation d'un bastion

- Identification

- c'est la phase où « on dit qui on est »
 - notion de login, username

- Authentification

- c'est la phase où « on prouve qu'on est qui on prétend être »
 - notion de secret partagé, password, ticket Kerberos

- L'identification peut être une première barrière pour un attaquant!

- Sous Linux, l'identification est gérée par la libnss, qui traite :

- les comptes locaux via le fichier /etc/password
 - les comptes distants via NIS ou LDAP

- On fait de l'authentification depuis longtemps :
 - techniques fondamentalement basées sur la présence physique des personnes :
 - témoignage d'un gardien
 - détention d'un badge
 - signature
 - repose essentiellement sur la dissuasion
 - contournement simple techniquement mais sévèrement réprimé
- L'authentification électronique a été conçue pour s'abstraire des limites des moyens classiques :
 - économie de surveillance
 - possibilité d'accès distants
 - disponibilité
- Le problème repose alors sur la notion de « preuve électronique »

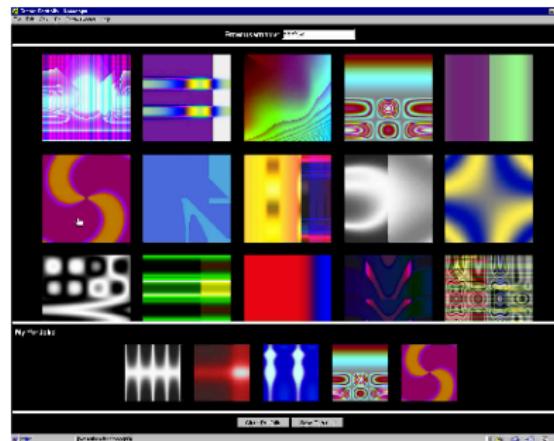
- Les protocoles d'authentification sont les méthodes utilisées pour mettre en place un contrôle d'accès logique aux systèmes d'informations (ici les systèmes Unix)
- Ces protocoles sont d'autant plus critiques que les utilisateurs aiment les systèmes « *Single Sign On* »
- Le contrôle d'accès se décompose en deux étapes :
 - l'identification : « Mon nom est Bond, James Bond »
 - l'authentification : « Voici mon code d'accès au MI6 »
- Plusieurs approches pour l'authentification sont possibles en fonction de ce que j'apporte pour m'authentifier :
 - quelque chose que je **connais**
 - quelque chose que je **possède**
 - quelque chose que je **incarne**

Un peu de théorie

- Quelque chose que je connais :
 - c'est l'approche la plus fréquemment rencontrée
 - l'utilisation d'un mot (phrase) de passe est l'implémentation la plus courante de cette approche
- On peut mentionner quelques alternatives connues :
 - implémentation par association
 - le système connaît préalablement des associations et vous en soumet une ou plusieurs pour validation
 - + : cela augmente la quantité d'information qu'un attaquant doit connaître pour usurper une identité
 - - : nécessite un arrangement compliqué utilisateurs / système
 - implémentation par « challenge-response » :
 - le système pose une question à laquelle seul l'agent identifié est sensé pouvoir répondre
 - schéma aussi utilisé dans l'approche (« ce que je possède »)

Un peu de théorie

- Usenix, Security Symposium 2000, Denver (CO), « Déjà-vu »
 - Rachna Dhamija
 - Problème du facteur humain dans l'authentification
 - Idée : facile à retenir, difficile à transmettre
 - Utilisation en production ?



- Quelque chose que je possède :
 - autre moyen de résoudre le problème de la transmission / duplication de l'authentifiant
 - après l'identification d'un agent par le système, un moyen de valider cette identité est de vérifier la possession par cet agent d'un dispositif que seule la personne identifiée possède
 - cela implique que :
 - système et agent se soient mis d'accord sur le dispositif à posséder
 - le système peut facilement et sûrement reconnaître ce dispositif
 - l'agent ne le perd / donne pas
- différentes implantations de cette approche sont possibles :
 - type « clef » : la possession du dispositif suffit (secureID)
 - type « carte à puce » : possession + mot de passe (code PIN)
 - type « calculette » : possession + challenge / response
- certains de ces mécanismes posent des problèmes de synchro
- résout partiellement certains pbs liés à l'approche par mot de passe (divulgation, trivialité, certaines attaques réseaux ... pas toutes)



- Quelque chose que j'incarne :

- cette approche consiste à vérifier l'identité prétendue par une caractéristique inhérente de l'agent identifié
 - chaque être humain possède plusieurs caractéristiques uniques : empreintes digitales, vocales, rétiniennes, forme de la main, ADN ...
 - résout certains problèmes des approches précédentes, notamment l'oubli, le don, la divulgation, etc.
 - rarement utilisé (coût, difficulté de mise en œuvre), uniquement dans des environnements où l'usurpation d'identité est très critique ou dans le cadre d'accès physiques
 - pose des problèmes juridiques, culturels, éthiques, morphologiques
 - répond plus au problème d'identification que d'authentification (il est difficile de changer d'empreinte rétinienne ...)

Authentification

Elle est dite « forte » si elle fait appel à **deux** facteurs de natures différentes.

- Deux facteurs parmi trois sont nécessaires :
 - je connais
 - je possède
 - j'incarne
- Approche privilégiée, on l'a dit, c'est le mix des deux premiers.
- Exemples :
 - accès SSH avec une clef protégée par une passphrase
 - accès graphique en Kerberos + carte à puce (PKINIT)
 - accès à une salle dont la clef est dans un coffre

■ Mécanisme d'authentification centralisé

- les applications « PAM-ifiées » délèguent leur authentification
- les bibliothèques PAM contrôlent cette authentification
- le paramétrage des bibliothèques PAM est effectué et contrôlé par l'administrateur pour chaque application
- la configuration est ventilée dans /etc/pam.conf et /etc/pam.d
- bien que répandu, c'est un mécanisme pourtant totalement optionnel : OpenBSD et Slackware ne l'intègrent pas, par exemple

- 4 groupes de gestion :

- **authentication** : vérification de l'identité (identifiant / authentifiant)
- **account** : vérification / définition des informations de compte (mot de passe expiré, appartenance à un groupe)
- **password** : mise à jour du mot de passe (niveau de robustesse, obliger l'utilisateur à changer de mot de passe après expiration)
- **session** : préparation de l'utilisation du compte (afficher un motd, tracer la connexion, monter des répertoires utilisateurs)

- 4 contrôles de réussite :

- **requisite** : tous ces modules doivent réussir, échec immédiat
- **required** : tous ces modules doivent réussir, échec retardé
- **sufficient** : la réussite de ce module suffit à valider la pile (le résultat est ignoré si un module requis a échoué avant)
- **optional** : module considéré uniquement s'il est seul en pile

■ Exemple : login

```
auth      required    /lib/security/pam_securetty.so
auth      requisite   /lib/security/pam_nologin.so
auth      required    /lib/security/pam_stack.so service=system-auth
    auth    required    /lib/security/pam_unix.so
    auth    optional   /lib/security/pam_permit.so
    auth    required    /lib/security/pam_env.so

account   required    /lib/security/pam_stack.so service=system-auth
    account required   /lib/security/pam_unix.so
    account optional   /lib/security/pam_permit.so
    account required   /lib/security/pam_time.so

password   required    /lib/security/pam_stack.so service=system-auth
    password required   /lib/security/pam_cracklib.so minlen=8 retry=3
    password sufficient /lib/security/pam_unix.so shadow sha512 nullok

session   required    /lib/security/pam_stack.so service=system-auth
    session required   /lib/security/pam_limits.so
    session required   /lib/security/pam_unix.so
    session optional   /lib/security/pam_motd.so
```

- Le mécanisme OPIE (One-time Passwords In Everything) est inclus dans FreeBSD et OpenBSD (entre autres)
 - il est supporté par `ftpd`, `login` et `su`
 - un module PAM le prend également en charge
- L'algorithme S/KEY est utilisé pour générer les mots de passe
 - lors de la phase de login, l'utilisateur reçoit un challenge
 - il doit recopier ce challenge dans une calculatrice
 - celle-ci fournit la réponse au challenge
 - l'utilisateur peut s'authentifier avec cette réponse
- Les commandes concernant l'usage d'OPIE :
 - `opiepasswd` : initialise OPIE pour un utilisateur donné
 - `opiekey` : calcule les réponses aux challenges
 - `opieinfo` : génère une liste de futures réponses

Autorisation

■ Objectifs de l'autorisation :

- l'authentification conditionne l'accès sur la preuve d'identité
- l'autorisation conditionne l'accès sur le besoin
 - besoin par rapport à un rôle particulier :
utilisateur, administrateur, client, fournisseur ...
 - besoin par rapport à une tâche particulière :
indexation, sauvegarde, exécution planifiée ...

■ Mécanismes d'autorisation :

- pas de mécanisme unifié sur les systèmes Unix
- traitement spécifique à chaque application
- la problématique semble en effet très complexe

■ Le terme autorisation est quasi-équivalent à celui de contrôle d'accès.

- Exemples de programme incluant du contrôle d'accès :
 - modules d'accès PAM (groupe « session »)
 - pam_access, pam_time
 - utilisé essentiellement par login
 - sudo
 - user ALL=(ALL) commands
 - sshd
 - {Allow|Deny}{Groups|Users}
 - apache
 - Directives Allow/Deny

■ Outils de surveillance immédiate

- sur les *file descriptor* : netstat, lsof, ...
- sur la mémoire : free, vmstat, top, ...
- sur les utilisateurs : getent, last, who, ...
- sur les permissions : audit2allow, ls, ...

■ Comptabilité (*accounting*)

- enregistrement des ressources utilisées par un processus

■ Accès pérénne à travers les journaux de logs

- pouvoir répondre à la question : « qu'est-ce qui s'est passé ? »
- par contre, il faut privilégier la qualité à la quantité
- problèmes : la collecte, la gestion, et l'extraction des informations

- syslogd (ou syslog-ng, ou rsyslog) est le daemon qui gère les logs système et noyau (éventuellement secondé par klogd)
- il permet la gestion des logs :
 - locaux, utilisation d'un socket Unix (/dev/log)
 - distants, écoute sur le port UDP/514
- les messages de logs sont taggés :
 - avec une catégorie (facility) :
auth, daemon, kern, syslog ...
 - avec un niveau d'importance (priority) :
debug, warning, alert, panic ...
- chaque message contient la date et le nom de l'hôte l'ayant émis
- /etc/syslog.conf permet de déterminer si le message doit être envoyé dans un fichier (et lequel), affiché en console, ignoré ...

- pour tester une configuration du daemon syslogd ou lui envoyer des messages depuis des scripts, on peut utiliser l'outil logger :
 - on spécifie une priorité et éventuellement une catégorie
 - on envoie un message au daemon
 - celui-ci le traite alors en fonction de ces paramètres, selon les indications du fichier de configuration
- pour faciliter l'exploitation simple des logs, on peut mentionner :
 - logrotate (Linux), newsyslog (BSD), multilog (daemontools)
 - permet l'archivage et la rotation des logs, en renommant et en compressant les anciens logs de façon automatique
 - lastlog, wtmp, utmp
 - la commande lastlog et les deux fichiers wtmp / utmp (commandes who, ac) permettent de voir qui est actuellement loggé sur le système, ou quand un utilisateur s'est connecté la dernière fois

- l'analyse des logs doit idéalement être effectuée régulièrement et avec attention, par un opérateur **humain**
- néanmoins, il s'agit d'un travail fastidieux, d'où l'existence d'outils permettant de faciliter cette exploitation
- les fonctionnalités couramment proposées sont :
 - stocker / indexer / redonner les logs
 - condenser / résumer / trier les logs
 - mettre en évidence certains logs
 - visuellement, via un système de consultation adapté
 - dans un rapport qui ne présente que les logs intéressants
 - effectuer un traitement périodique des logs
 - résumé journalier envoyé par mail à l'administrateur, par exemple
 - effectuer un traitement sur une durée déterminée
 - exécuter une commande donnée lorsqu'un type de log est rencontré
- cela ne doit pas se substituer à une conservation traditionnelle

- il peut être intéressant de ne pas se contenter de loguer les débuts et fins de sessions des utilisateurs, mais aussi les commandes exécutées
 - dans le cas où l'utilisateur paie pour utiliser le système, en fonction des ressources qu'il consomme
 - dans le cas où l'on veut surveiller l'activité des utilisateurs, les programmes qu'ils utilisent (exemple : john) ...
- un support de cette fonctionnalité est nécessaire au niveau du noyau
- les informations obtenues sont de type : utilisateur, commande, conditions d'exécution, de terminaison, date de fin, temps, ...
- les données sont sauvegardées dans un fichier (accès à restreindre)
- les outils associés :
 - accton : active / désactive l'accounting
 - sa : résumé de l'accounting
 - lastcomm : infos sur une commande passée

- 1** Préambule
 - Sécurité physique
- 2** Sécurité système
 - Authentification
 - Autorisation
 - Audit
- 3** Sécurité applicative
 - Menaces
 - Contre-mesures
- 4** Installation d'un bastion

→ On peut les classer en fonction de la cause :

- **hardware :**

- sensibilité humidité / poussière
- stockage non protégé

- **software :**

- tests insuffisants
- manquements lors des audits

- **réseau :**

- liaisons non sécurisées

- **personnel :**

- recrutement inadapté
- mauvaise sensibilisation

- **site :**

- zone sujette aux inondations
- source de courant non fiable

- **organisationnel :**

- manque d'audits réguliers

Les menaces sur les applications

- social engineering (« c'est michel, de l'informatique »)
- buffer overflow (heap, stack, ...)
- format string (« je m'appelle %s »)
- injection SQL (« SELECT usr WHERE pwd = '' OR 1 = 1 »)
- injection de code (« eval(\$userinput); »)
- directory traversal (« cat ../../../../../../etc/shadow »)
- XSS (« je m'appelle <script>alert() ;</script> »)
- race conditions (systèmes, applicatives, ...)
- CSRF (« »)
- clickjacking (principalement pour générer des revenus)
- privilege escalation (« can i haz r00t plz? :) »)
- *blame the victim*, abus de confiance, culpabilisation

Les menaces sur les applications

- social engineering (« c'est michel, de l'informatique »)
- buffer overflow (heap, stack, ...)
- format string (« je m'appelle %s »)
- injection SQL (« SELECT usr WHERE pwd = '' OR 1 = 1 »)
- injection de code (« eval(\$userinput); »)
- directory traversal (« cat ../../../../../../etc/shadow »)
- XSS (« je m'appelle <script>alert() ;</script> »)
- race conditions (systèmes, applicatives, ...)
- CSRF (« »)
- clickjacking (principalement pour générer des revenus)
- privilege escalation (« can i haz r00t plz? :) »)
- *blame the victim*, abus de confiance, culpabilisation

Les menaces sur les applications

- quelles sont les principales causes des vulnérabilités publiées chaque jour, dans les bulletins de sécurité ?
 - principalement des erreurs de programmation
 - mauvais contrôle des données fournies par un utilisateur
 - oubli de cas particuliers (environnement, logique, ...)
 - et plus rarement, de la malveillance discrète
 - aussi bien dans des applications type serveur (ftpd, bind, sendmail, ...) que dans des applications clientes (du web notamment, avec les moteurs JS, Java, Flash, et les CMS)
- les possibilités d'exploitation sont alors multiples :
 - atteinte à la **disponibilité** : DoS (Déni de Service), plantage
 - atteinte à la **confidentialité** : accès à des données privées
 - atteinte à l'**intégrité** : modification de données sensibles

Les menaces sur les applications

- protection des applications :
 - cloisonnement applicatif
 - chroot jail
 - docker
 - machines virtuelles
 - cloisonnement noyau
 - grsecurity
 - SELinux
 - AppArmor
- amélioration des procédés
- problème : on ne peut pas patcher les gens ...

Les attaques de type *format strings* reposent sur le principe suivant :

- profiter du manque de précautions dans la programmation d'une appli
 - pas ou peu de contrôle des données passées par l'utilisateur
 - utilisation de fonctions avec des chaînes de format : `printf`, ...
 - où la chaîne de format elle-même est passée par l'utilisateur
- passer ainsi une chaîne de caractères spéciale à l'application
 - contenant une chaîne de format spécialement construite
 - permettant de lire ou écrire en mémoire des informations
- on peut soit récupérer des informations cruciales (comme l'offset de la section de code suite au décalage par l'ASLR), soit écrire un octet choisi à une adresse (qu'on peut éventuellement contrôler)

- code :

```
1 int main(int argc, char** argv)
2 {
3     char donnees[7] = "secret";
4     printf("premier argument :\n");
5     printf(argv[1]);
6     return 0;
7 }
```

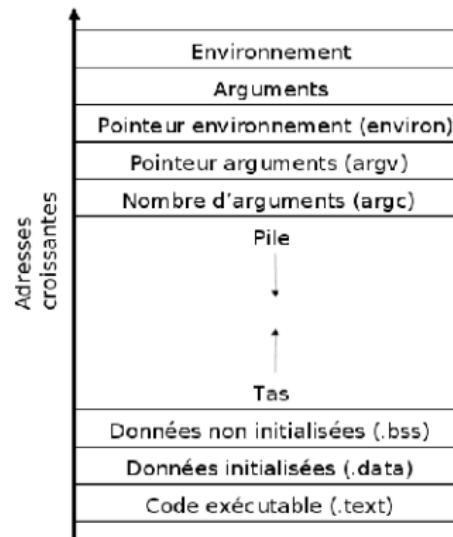
- si argv[1] contient une chaîne de format, on peut dépiler autant d'items que l'on veut, sous la forme que l'on veut :
 - %s : une chaîne
 - %p : une adresse
 - %d : un entier
- avec %n, on peut même écrire à l'adresse pointée par la pile le nombre d'octets déjà écrits (*Write Anything Anywhere*)

Les attaques de type *buffer overflow* reposent sur le principe suivant :

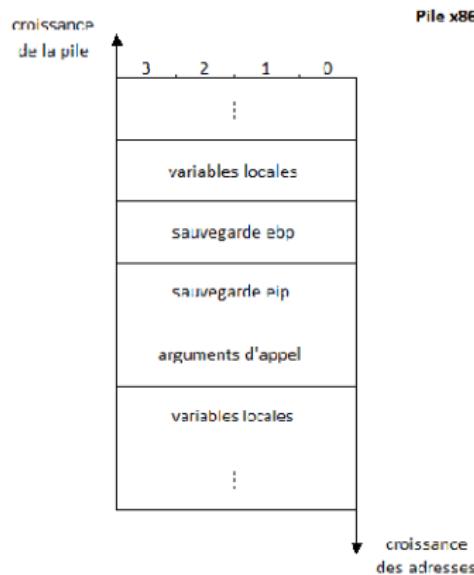
- profiter du manque de précaution dans la programmation d'une application
 - pas ou peu de contrôle des données passées par l'utilisateur
 - utilisation de fonctions dangereuses : `strcpy`, `sprintf`, `gets`, ...
- passer ainsi une chaîne de caractères spéciale à l'application
 - contenant du code exécutable
 - permettant de dérouter le flux d'exécution vers ce code
- faire exécuter par l'application le code construit par l'attaquant (avec, notamment, les droits de l'application en question)

Buffer Overflows

- représentation simplifiée de la mémoire virtuelle d'un processus :



- représentation simplifiée de la structure d'une pile sur x86 :



Buffer Overflows

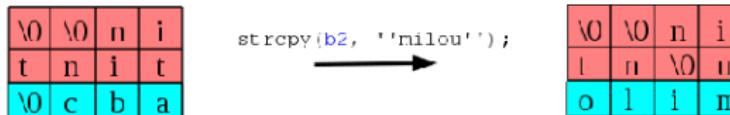
- vulnérabilités liées à la gestion des buffers :

- code :

```
1 int main(int argc, char** argv)
2 {
3     char b1[7] = "tintin";
4     char b2[4] = "abc";
5     strcpy(b2, argv[1]);
6     printf("b1 : %s\nb2 : %s\n", b1, b2);
7     return 0;
8 }
```

- en mémoire, dans la pile (exécution de "../prog milou")

- appel de `strcpy`



- appel de `printf`

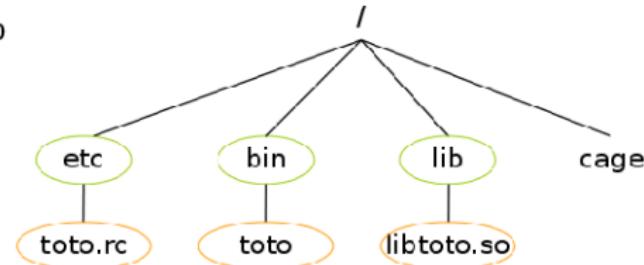
- la possibilité d'écraser des données dans la pile permet dans certains cas de faire exécuter du code arbitraire au programme vulnérable :
 - on utilise un *shellcode*
 - appelé ainsi car il permet usuellement d'obtenir un shell
 - il consiste en une suite d'instructions assembleur passée sous forme de chaîne de caractères
 - le but du jeu est de modifier, en écrasant sa valeur dans la pile, l'adresse de la prochaine instruction à exécuter (registre RIP)
 - l'idée est de la faire pointer vers les instructions de notre code
- ici il s'agit d'exploiter simplement des données de la pile, mais il y a des variantes (*heap overflow*, *heap spraying*, *return-into-libc*, ...)
- l'intérêt de faire exécuter du code personnalisé à un tel programme se manifeste quand celui-ci est Set-UID (notamment root)

- les types d'attaques que nous avons évoqués découlent de techniques de programmation trop permissives ou oublieuses des problématiques de sécurité
 - une programmation rigoureuse et consciente de ce genre de problèmes permet d'éviter une première vague de vulnérabilités
- mais on ne peut avoir une totale confiance dans les applications que l'on utilise, d'où le besoin de protections supplémentaires au niveau système (« approche ceinture et bretelles »)
- intéressons-nous alors à des solutions qui, déployées sur le système, permettent de limiter ou d'empêcher l'exploitation des failles qui ne manqueront pas de se manifester sur notre SI

- s'il est capable d'exploiter une faille d'un service, un pirate va chercher à exécuter d'autres commandes sur la machine
- une parade consiste à limiter l'environnement du daemon
- le principe de la cage :
 - dissimuler une partie du système de fichiers au programme
 - ne lui laisser voir **que** ce dont il a besoin pour s'exécuter correctement
 - fichiers de configuration
 - bibliothèques dynamiques
 - autres exécutables, scripts
 - fichiers spéciaux (*pipes, sockets, ...*)
 - fichiers de log
- en pratique on fait passer un répertoire du système pour la racine (/), l'application ne peut (en théorie) pas voir en dehors de ce répertoire

- exemple de mise en cage avec chroot :
 - fichier exécutable : /bin/toto
 - configuration : /etc/toto.rc
 - bibliothèque : /lib/libtoto.so
 - nouvelle racine : /cage

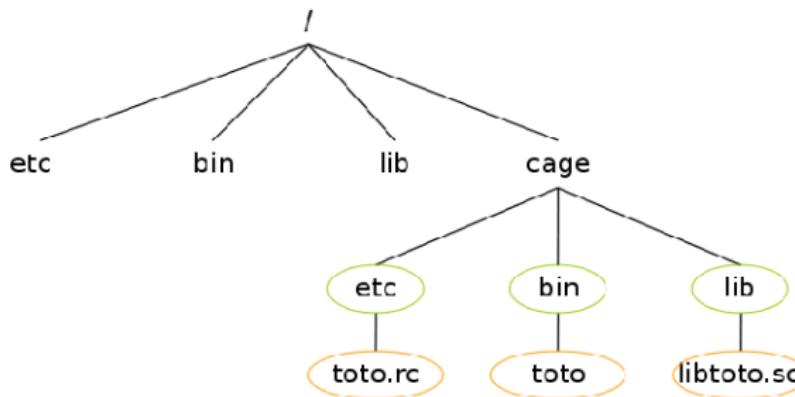
- on reproduit l'arborescence sous /cage :
 - mkdir /cage/etc
 - mkdir /cage/bin
 - mkdir /cage/lib



Construction d'une cage

- on copie les fichiers du daemon :

- cp /bin/toto /cage/bin/toto
- cp /etc/toto.rc /cage/etc/toto.rc
- cp /lib/toto.so /cage/lib/libtoto.so



Construction d'une cage

- on lance ensuite la commande : chroot /cage /bin/toto
- comment déterminer les fichiers nécessaires au programme et devant se trouver dans la cage ?
 - on peut distinguer deux types de fichiers :
 - les bibliothèques dynamiques
 - le reste
 - **certaines** des bibliothèques dynamiques sont données par ldd
 - on copie ces bibliothèques dans l'arborescence de la cage
 - si on dispose des sources, on peut éviter ce problème en recompilant l'exécutable statiquement (édition de liens)
 - les divers fichiers accédés par le programme lors de son exécution peuvent être déterminés avec un outil tel que strace
- certains daemons ont des options de lancement pour exécuter automatiquement un appel à chroot. attention au paradoxe !

Séparation des privilèges

- autre problématique : un programme ayant besoin des droits *root* à un moment de son exécution doit-il pour autant les conserver tout au long de son exécution ?
 - cas typiques : tcpdump, serveurs sur un port < à 1024, etc.
- la séparation de privilèges propose une solution :
 - un ou des utilisateurs spécifiques sont créés pour l'application
 - celle-ci est lancée avec les droits *root*
 - les opérations nécessitant ces droits sont effectuées en tant que *root*
 - les autres opérations sont effectuées avec les droits de l'utilisateur spécifique (changement d'*uid / gid*)
 - pour cela, utilisation de *fork* :
 - exécution du processus père avec des droits privilégiés
 - exécution du processus fils avec des droits restreints
- exemples : sshd, vsftpd, popa3d, qmail

- dans le cas de la séparation de priviléges, on crée des utilisateurs spécifiques, qui ne peuvent pas se connecter
- il existe une autre catégorie d'utilisateurs spécifiques, voués à l'exécution d'une tâche unique, ce sont les comptes dits **captifs** :
 - leur shell n'est pas un interpréteur de commande usuel.
attention à ce que l'application choisie ne permette pas l'exécution de commandes non contrôlées ! (comme more par exemple)
 - lorsqu'ils se connectent, l'application définie comme shell s'exécute
 - l'exécution terminée, l'utilisateur est déconnecté sans avoir eu la main directement sur le système
- cette fonctionnalité peut être utilisée pour sécuriser le système, bien qu'historiquement ce ne soit pas ce qui a motivé sa création
- quelques exemples :

```
date::60000:100:Run the date program:/tmp:/usr/bin/date
uptime::60001:100:Run the uptime program:/tmp:/usr/bin/uptime
```

- on peut limiter les fonctionnalités de l'interpréteur de commande de certains utilisateurs (comptes ouverts ou utilisateurs spécifiques)
- pour cela, un shell restreint permet notamment :
 - de restreindre un utilisateur au répertoire courant (\$HOME)
 - de l'empêcher de modifier son \$PATH
 - de l'empêcher d'exécuter des commandes absentes de son \$PATH
 - de l'empêcher de modifier des variables d'environnement
 - de limiter les redirections d'entrées / sorties (>, >>)
- les variables d'environnement sont fixées par un fichier de configuration (typiquement .profile) au login, et ne peuvent ensuite être modifiées
- des shells tels que bash, zsh ou ksh proposent un mode restreint

- Pour lutter contre la majorité des *buffer overflows*
 - pile et/ou tas non exécutables : les zones mémoire accessibles en écriture ne sont pas exécutables
 - adresses aléatoires des fonctions de bibliothèques dynamiques : gêne les attaques de type *return-into-libc*
 - protections au moment de la compilation : empêcher l'écrasement de certaines informations-clé de la pile (*canary / stack guard*)
 - réimplémentation des fonctions dangereuses de la libc (*libsafe*)
- Pour limiter les possibilités d'exploitation d'une faille
 - chroot permet de restreindre ce qui est perçu, et donc accessible, au niveau du système de fichiers, par l'application vulnérable
 - MAC (Mandatory Access Control)
 - définition de politiques de sécurité pour le contrôle d'accès
 - contrôle d'accès obligatoire par un mécanisme central
 - peut favoriser une plus grande granularité au niveau de la gestion des différents accès (par exemple : contrôler les appels systèmes)

- 1** Préambule
 - Sécurité physique
- 2** Sécurité système
 - Authentification
 - Autorisation
 - Audit
- 3** Sécurité applicative
 - Menaces
 - Contre-mesures
- 4** Installation d'un bastion

Installation d'un bastion

- objectifs :

- planification de l'installation
 - pourquoi minimiser le système ?

- noyau :

- être ou ne pas être modulaire
 - le choix des pilotes

- partitions :

- le bon partitionnement conditionne la vie du système

- applications :

- bien choisir les packages

Planification de l'installation

- *roadmap*
- objectif
 - construire un système sécurisé selon ses besoins
- avant l'installation
 - choisir une distribution, vérifier l'intégrité des données
- pendant l'installation
 - partitionnement, « *simple is beautiful* »
- après l'installation
 - configuration, du système jusqu'au réseau

Planification de l'installation

- choix d'une distribution Unix
 - orientés RPM : RedHat, CentOS, Mandrake, Suse, Fedora
 - **pour** : présence de support pour les entreprises
 - **contre** : un peu en retard, patches « maison »
 - orientés APT : Debian, Ubuntu
 - **pour** : très fonctionnel, documentation complète
 - **contre** : *bleeding edge*, à destination des développeurs
 - orientés sources : Linux From Scratch, Gentoo
 - **pour** : construit selon ses besoins, très minimaliste
 - **contre** : long à construire, à configurer, donc à utiliser

Planification de l'installation

- choix d'une distribution Unix
 - ArchLinux
 - **pour** : *rolling release*, toujours à jour, système et paquets
 - **contre** : la mise à jour constante oblige une administration réactive
 - Slackware
 - **pour** : fichiers de configuration non modifiés, *up-to-date*
 - **contre** : système un peu trop simpliste, un peu « brut »
 - autres, à usages spécifiques (BlackArch, ClipOS, ...)
 - **pour** : très efficaces dans leur domaine / raison de conception
 - **contre** : difficilement réutilisables pour un autre motif si besoin

Planification de l'installation

- choix d'un système *BSD
 - FreeBSD : orienté performance et compatibilité
 - **pour** : très performant, sécurisation forte possible
 - **contre** : tout à faire « à la main » : mise à jour, sécurisation, le moins d'architectures supportées
 - OpenBSD : orienté simplicité et sécurité
 - **pour** : audit et amélioration du code source des applications, tout est chrooté, pile non exécutable, minimalisme assez radical
 - **contre** : moins performant, très peu orienté station de travail
 - NetBSD : orienté portabilité et rétro-compatibilité
 - **pour** : un nombre record de plateformes supportées
 - **contre** : pas aussi sûr que OpenBSD, pas aussi performant que FreeBSD, mais il tourne sur tout !

Planification de l'installation

- préparation de l'installation et mises à jour
 - déroulement idéal
 - récupérer toutes les mises à jour disponibles
 - installer la machine hors réseau, présence de failles tant que l'OS n'est pas à jour
 - vérifier l'intégrité
 - à vérifier après chaque téléchargement, sur chaque CD
 - ne fournit aucune garantie quant à l'origine, recalcul des hashs par le pirate possible
 - vérifier les signatures numériques
 - garantit l'origine, l'intégrité, ...
mais ne garantit pas forcément l'absence de problèmes ;-)

Pourquoi minimiser le système ?

- pourquoi ne pas faire une installation classique,
et n'utiliser que les outils dont on a besoin ?
 - occupation inutile de l'espace disque
 - plus d'applications : plus de mises à jour à effectuer
(patches de sécurité notamment)
 - plus d'applications : plus de failles potentielles
 - plus d'applications : probabilité plus forte pour un intrus
de trouver des outils lui facilitant la tâche
 - un outil suspect sera plus facilement repéré s'il n'est pas
noyé dans la masse

Pourquoi minimiser le système ?

- limiter les outils installés et les services actifs au strict nécessaire
- procéder de même, si possible, pour les fonctionnalités du noyau
- de façon générale, supprimer le superflu permet :
 - de diminuer la surface d'attaque (limiter les failles)
 - de repérer plus facilement un élément suspect
- cela s'applique aux fonctionnalités du noyau :
 - la gestion de certains périphériques ou systèmes de fichiers peut s'avérer plus dangereuse qu'utile (USB par exemple)
 - un noyau monolithique sans support des modules permettra d'éviter le chargement de modules piégés
 - le noyau est la base du système, il est primordial que cette base soit saine (rappelez-vous de l'effet « court-circuit »)

Pourquoi minimiser le système ?

- dans le cas des services :
 - ouvrir des services inutiles peut permettre à un attaquant :
 - d'obtenir des informations (OS fingerprint, uptime ...)
 - de pénétrer plus facilement dans le système
 - de le faire tomber avec un DoS (exemple : echo chargen)
 - un port ouvert mais non autorisé sera également mieux détecté (ICMP admin prohibited par exemple)
 - laisser ces services actifs mais en interdire l'utilisation via du filtrage réseau est une alternative, moins propre que l'absence du service
- distinction entre service public et service d'administration
 - le monde extérieur ne doit pas avoir accès aux services d'admin
 - n'écouter que sur l'interface adéquate
 - filtrer les paquets réseau en entrée
 - restreindre les applications et usages

- quelques éléments ne sont pas forcément les bienvenus pour un noyau que l'on souhaite sûr
 - la gestion des modules peut s'avérer dangereuse : pour un serveur on favorisera un noyau monolithique (plus gros)
 - connaître son système permet de savoir ce dont il n'a pas besoin :
 - inutile de supporter tous les systèmes de fichiers, toutes les cartes graphiques, tous les protocoles réseau, tous les firmwares
 - de même, on peut sciemment inhiber certains périphériques ou certaines fonctions en ne les gérant pas dans le noyau : USB DiskOnKey, claviers, lecteurs / graveurs externes, port série, port parallèle, ou pour le réseau : ipv6, ppp, 802.11, infrarouge, bluetooth
- le bon sens est notre ami!
 - un serveur aura rarement besoin de pilote pour sa carte son ou WiFi
 - idem pour les fonctionnalités expérimentales

Noyau : les pilotes

- le minimum vital pour un noyau ...
 - sont indispensables pour le bon fonctionnement du système :
 - le support des périphériques (disques SCSI, cartes réseau ...)
 - le support des systèmes de fichiers (ext3 ? ramfs ?)
 - les protocoles réseau usuels
 - certains systèmes d'exploitation offrent des fonctionnalités axées sécurité qui peuvent s'avérer utiles, voire nécessaires :
 - un pare-feu (`netfilter`, `pf`, ...)
 - des algorithmes cryptographiques (TLS, chiffrement de partitions)
 - une implémentation d'IPsec
 - une gestion des politiques de sécurité
 - des protections système spécifiques (pile non exécutable d'OpenBSD et grsecurity, contraintes sur certains fichiers)

Applications nécessaires

- le kit de survie doit contenir :
 - les commandes Unix de base (compilées en statique si possible)
 - gestion des processus : ps, top, kill, ...
 - simples outils texte : cat, more, ...
 - manipulation de fichiers : cp, rm, mv, ls, ...
 - un éditeur de texte, vi le plus souvent
 - un interpréteur de commandes (shell) : sh, bash, csh, zsh, ...
- l'administration de la machine a aussi ses nécessités :
 - outils de configuration réseau : ifconfig, netstat, route
 - outils de configuration du système : gestion du système de fichiers, des utilisateurs, des paquets, shutdown, ...
 - serveur sshd si l'accès par le réseau est autorisé
 - outils d'administration de sécurité, en fonction de l'installation :
 - paramétrage du pare-feu (iptables, pfctl, ipfw, ...)
 - configuration IPsec (ipsecctl, sasyncd, isakmpd, ...)

Applications nécessaires

- n'oublions pas le principal !
 - s'il s'agit d'un serveur, des applicatifs spécifiques seront installés
 - même si les précautions évoquées jusqu'ici pourront s'appliquer au niveau de sa configuration, on entre dans le domaine de la sécurité au niveau applicatif plus que système
- quelques exemples simples :
 - serveur HTTP (typiquement, Apache)
 - architecture modulaire : choix rigoureux des *plugins* installés
 - accès aux répertoires, autorisation de lister les contenus, ...
 - serveur FTP
 - limitation des commandes autorisées
 - éviter les serveurs versant dans la surenchère de fonctionnalités
- la sécurisation de ces daemons fait intervenir d'autres concepts abordés plus loin (au niveau confinement / paramétrage applicatif)

Applications superflues

- laissons de côté le confort moderne pour plus de sécurité
 - certains éléments sont souvent nécessaires lors de l'installation et de la configuration initiale de la machine, mais représentent un réel danger en production :
 - compilateurs et outils associés : cc, gcc, make
 - debuggers, désassemblateurs : gdb, valgrind
 - outils d'analyse de binaire : objdump, readelf
- l'installation de X11 se justifie rarement sur un serveur
(à moins bien sûr d'être nécessité par un service fourni : TS)
- de manière générale, éviter les outils qui permettraient à un intrus de recueillir des informations variées si leur présence n'est pas justifiée :
 - tcpdump, lsof, nc, netcat, ...

- des cas difficiles à trancher ...
 - certains outils occasionnellement utiles à un administrateur pourront l'être encore plus pour un intrus
 - les applications d'administration centralisée (puppet)
 - les modules et scripts d'audit de la sécurité !
 - les langages de scripts (perl, python, ...)
 - pratiques pour réaliser des tâches courantes (scripts d'administration)
 - adaptés également pour des petits programmes à utilisation ponctuelle
 - mais véritables boîtes à outils aux possibilités immenses (perl a été qualifié de *Unix's Swiss Army Chainsaw !*)
 - certains langages permettent d'obtenir des exécutables compilés
- il n'y a pas de « recette » absolue
 - une bonne connaissance de son système, un zeste de bon sens et beaucoup de paranoïa permettent d'obtenir un degré de sécurité relativement satisfaisant (attention aux excès de confiance)

- une fois qu'il a été décidé de ce qui doit ou non être présent sur le bastion, place à l'installation
- le procédé sera fortement dépendant de :
 - l'OS choisi
 - la distribution adoptée
 - ces choix peuvent découler de contraintes liées au contexte
 - contraintes **matérielles** : l'OS choisi permet-il la pleine exploitation du matériel de la machine (RAID, SCSI, multi-processeurs, ...)?
 - contraintes **d'installation** : la distribution choisie permet-elle une installation sur une machine avec seulement 256Mo de RAM ?
 - contraintes **contractuelles** : souhaite-t-on utiliser une application certifiée pour un OS ou une distribution ? problème de licence ?

- il est assez peu probable d'avoir dès l'installation tous les packages qu'on veut, et seulement eux. on peut considérer 3 grandes familles de systèmes d'exploitation ou de distributions :
 - les systèmes sobres, permettant de faire une installation minimale, et d'ajouter par la suite les packages que l'on souhaite (OpenBSD)
 - les systèmes lourds, installant de nombreux outils qui devront être supprimés par la suite (Redhat, SuSe, Mandrake)
 - les systèmes intermédiaires, où il est possible de choisir précisément ses packages dès l'installation du système (Debian, FreeBSD)
- parfois il est possible de choisir des méta-packages : ensembles de packages regroupés par thèmes (environnement graphique, réseau)
- selon le type d'installation choisi, on devra ensuite épurer le système ou au contraire le compléter (et c'est préférable : on sait ce qu'on a)

Commissariat à l'énergie atomique et aux énergies alternatives
Centre de Bruyères-le-Châtel | 91297 Arpajon Cedex
T. +33 (0)1 69 26 40 00 | F. +33 (0)1 69 26 40 00
Établissement public à caractère industriel et commercial
RCS Paris B 775 685 019

CEA DAM
DSSI
CTSI