M2-TIW4 sécurité des systèmes d'informations
Examen – durée 1h30

Seuls les supports non-électroniques sont autorisés. La concision, la précision et la clarté des réponses font partie des critères d'évaluation. Le barème est indicatif.

Exercice 1 : Questionnaire (réponses ouvertes mais courtes) (/6)

1. Définir ce qu'est le risque en sécurité. Donner les principales possibilités de traitement de ces risques.
2. Résumer les principaux avantages et inconvénients des méthodes de chiffrement symétriques et asymétriques.
3. Définir ce que sont les attaques *brute force*, au dictionnaire et par *rainbow tables*.
4. Définir les principes du moindre privilège et de séparation des tâches.
5. Résumer les principaux avantages des modèles RBAC.
6. Expliquer ce qu'est un *buffer-overflow*.

Exercice 2 : Problème de modélisation des droits (/8)

Il s'agit de définir *formellement* un modèle de contrôle d'accès spécifique. Ce modèle a pour objectif de protéger la vie privée et de garantir la sécurité de données médicales du Dossier Médico-social Partagé (DMSP). Les informations médicales du DMSP sont représentées par des *événements*, stockés dans un SGBD relationnel. On ne s'intéresse *qu*'à la lecture. Chaque événement a un *auteur* ainsi qu'un *formulaire* associé. Un auteur peut toujours *lire* un événement qu'il a produit lui-même.

Pour faciliter la définition et la gestion des accès à l'information du patient, on souhaite définir un modèle de contrôle d'accès discrétionnaire spécifique, organisé selon la notion d'*épisode*. Un épisode est sous-ensembles d'événements du dossier. Chaque événement est associé à *au plus un épisode*. On souhaite que le propriétaire puisse associer à chaque épisode des classes de confiance d'utilisateurs. La classe détermine qui peut lire les événements et si les événements produits par un utilisateur sont lisibles. Trois classes ont été identifiées :

Exclusive l'utilisateur ne voit dans l'épisode que les événements qu'il a produit lui-même et les événements qu'il produit ne sont visibles que par lui-même et le propriétaire ;

Restricted l'utilisateur voit les événements de l'épisode autorisés par son rôle mais les événements qu'il produit ne sont visibles que par lui-même et le propriétaire ;

Full l'utilisateur voit les événements de l'épisode autorisés par son rôle et les événements qu'il produit sont visibles par tous les utilisateurs ayant une relation de confiance pour cet épisode, en fonction de leur rôle ;

La gestion des permissions de bases est effectuée selon un modèle RBAC où on considère l'accès aux formulaires comme des permissions (autrement dit, on a une relation *Ura* entre utilisateurs et rôles et une relation *Pra* entre rôles et formulaires. Les rôles ne sont pas hiérarchisés.) Il s'agit de *réduire* ces permissions avec les classes de confidentialité ci-dessus. Le modèle à définir est donc un modèle de masquage qui vient se superposer à la gestion de base RBAC.

1. Identifier les principaux ensembles de ce modèle et les principales fonctions ou relations formelles qui les lient.
2. D'après les définitions des classes *Exclusive*, *Restricted* et *Full*, laquelle « manque » ? Donner sa sémantique informelle.
3. On définit formellement les classes comme des fonctions (XX pour *Exclusive*, SX pour *Restricted*, SS pour *Full* et XS pour celle de la question précédente) qui associent des utilisateurs aux épisodes. Donner le type formel de ces fonctions et préciser les conditions qu'elles doivent satisfaire.
4. Définir formellement une relation $RBAC$ entre utilisateurs et formulaire qui indique si l'accès est autorisé selon RBAC et deux fonctions :
   – *lecteur* associe à un épisode les utilisateurs qui peuvent lire ses événements ;
   – *invisible* associe à un épisode les utilisateurs dont les productions seront cachées ;
5. On note par U l'ensemble des utilisateurs, Ev celui des événements et Ep celui des épisodes. Définir formellement une fonction $auth : U \times Ev \to \{tt, ff\}$ qui à un événement et un utilisateur indique si ce dernier a le droit de lecture. On peut utiliser une fonction auxiliaire $aux : U \times Ep \times U \to \{tt, ff\}$ entre accédant, épisode et auteur.
6. La partie RBAC est formellement représentée par la fonction $authRBAC : U \times Ev \to \{tt, ff\}$. Définir formellement cette fonction ainsi que la fonction d'accès « finale » *access* après masquage selon les classes.
7. On s'intéresse aux droits sur un certain dossier composé de 7 événements (e_1 à e_7). Formaliser les assertions suivantes et calculer la matrice des droits résultante :
   – *Guru* est dans la classe *Exclusive*, *MyPhys* et *MyNurse* sont dans la classe *Full* de l'épisode E_1 ;
   – *MyPhys* et *OtherPhy* sont dans la classe *Restricted* et *MyNurse* dans *Full* de l'épisode E_2 ;
   – *Guru*, *MyPhys* et *OtherPhy* sont affectés au rôle *Physician*, *MyNurse* au rôle *Nurse* ;
   – le rôle *Physician* peut accéder aux formulaires *General* et *Treatment*, le rôle *Nurse* peut seulement accéder à *General* ;
   – l'événement e_1 est écrit par *MyNurse*, il est associé au formulaire *General* et n'appartient à aucun épisode ;
   – l'événement e_2 est écrit par *MyPhys*, il est associé au formulaire *Treatment* et n'appartient à aucun épisode ;
   – l'événement e_3 est écrit par *MyPhys*, il est associé au formulaire *General* et appartient à l'épisode E_1 ;
   – l'événement e_4 est écrit par *Guru*, il est associé au formulaire *Treatment* et appartient à l'épisode E_1 ;
   – l'événement e_5 est écrit par *MyPhys*, il est associé au formulaire *Treatment* et appartient à l'épisode E_2 ;
   – l'événement e_6 est écrit par *MyPhys*, il est associé au formulaire *General* et appartient à l'épisode E_2 ;
   – l'événement e_7 est écrit par *OtherPhy*, il est associé au formulaire *General* et appartient à l'épisode E_2 ;

Ce modèle est proposé dans : T. Allard, N. Anciaux, L. Bouganim, P. Pucheral, R. Thion. « Seamless access to health-care folders with strong privacy guarantees ». *Journal of Healthcare Delivery Reform Initiatives, 1(4) :82–107, 2009.*

Exercice 3 : Injection SQL (/2)

Voici un extrait de code C# s'appuyant sur une base de données relationnelle où l'utilisateur remplit le champ `ItemName` d'un formulaire. Le résultat est filtré selon l'identité de l'accédant :

```
string userName =ctx.getAuthenticatedUserName();
string query = "SELECT * FROM items WHERE owner = '"
               + userName + "' AND itemname = '" + ItemName.Text + "'";
```

```
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);
```

1. Supposons que l'utilisateur saisisse « `name' OR 'a'='a` », quel sera l'effet ?
2. Supposons que l'utilisateur saisisse « `name'; DELETE FROM items; --` », quel sera l'effet ? Pourquoi cette chaîne se termine par `--` ?
3. Donner des exemples d'impacts qu'une injection SQL peut avoir.
4. Donner des exemples de techniques pour se prémunir contre de telles attaques.

Exercice 4 : Analyse du bulletin MS10-092 (/3)

Voici le début du bulletin MS10-092 :

> *This security update resolves a publicly disclosed vulnerability in Windows Task Scheduler. The vulnerability could allow elevation of privilege if an attacker logged on to an affected system and ran a specially crafted application. An attacker must have valid logon credentials and be able to log on locally to exploit this vulnerability. The vulnerability could not be exploited remotely or by anonymous users.*
>
> *[...]*
>
> *An elevation of privilege vulnerability exists in the way that the Windows Task Scheduler improperly validates whether scheduled tasks run within the intended security context. An attacker who successfully exploited this vulnerability could run arbitrary code in the security context of the local system. An attacker could then install programs ; view, change, or delete data ; or create new accounts with full user rights.*

Les jobs du Task Scheduler sont décrits par des fichiers xml de la forme suivante. Notons qu'il est possible d'ajouter des commentaires dans ce fichier avec les balises `<!-- comment -->`. Le dossier où sont stockés ces fichiers est lisible par LocalSystem et les administrateurs locaux. Un utilisateur (sauf guest) peut écrire dans ce dossier. Pour protéger l'intégrité des commandes (qui sont exécutée par le Task Scheduler qui est un utilisateur privilégié), un checksum est calculé à la création de la tâche. Quand le job est lancé, le checksum est recalculé et comparé à celui enregistré avant d'être exécuté. Un algorithme de *cyclic redundancy check* (CRC) est utilisé pour cela (en l'espèce, CRC32).

```
<Principals>
  <Principal id="LocalSystem">
    <UserId>S-1-5-18</UserId>
    <RunLevel>HighestAvailable</RunLevel>
  </Principal>
</Principals>
<Actions Context="LocalSystem">
  <Exec>
    <Command>C:\WINDOWS\NOTEPAD.EXE</Command>
    <Arguments></Arguments>
  </Exec>
</Actions>
```

Wikipedia définit un CRC comme :

> *un outil logiciel permettant de détecter les erreurs de transmission ou de transfert par ajout, combinaison et comparaison de données redondantes, obtenues grâce à une procédure de hachage. Ainsi, une erreur de redondance cyclique peut survenir lors de la copie d'un support (disque dur, CD-Rom, DVD-Rom, clé USB, etc...) vers un autre support de sauvegarde.*

Les CRC sont évalués (échantillonnés) avant et après la transmission ou le transfert, puis comparés pour s'assurer que les données sont strictement identiques. Les calculs de CRC les plus utilisés sont conçus afin de pouvoir toujours détecter les erreurs de certains types, comme celles dues par exemple, aux interférences lors de la transmission.

La description de l'exploit est la suivante :

1. *Create a job that will be run under the current user account with the least available privileges ;*
2. *Read the task configuration file corresponding to the task created at step 1 and calculate its CRC32 checksum ;*
3. *Modify the task configuration file corresponding to the task created at step 1 so that it matches the same check sum as the original file and set the following properties :*
   (a) *Run the task.Principal Id=LocalSystem (principal for the task that provides security credentials) ;*
   (b) *Run the task.UserId=S-1-5-18 (SID of the LocalSystem) ;*
   (c) *Run the task.RunLevel=HighestAvailable (run with the highest available privileges) ;*
   (d) *Run the task.Actions Context=LocalSystem (security context under which the actions of the task are performed) ;*
4. *Run the task.*

1. Dans quelle catégorie (par exemple, selon la classification SANS) classeriez-vous cet exploit ?
2. Cet exploit est un des 0-days de Stuxnet. Expliquer de quoi il s'agit.
3. Quelle est le problème qui rend l'étape 3 de cet exploit possible ? De quelle forme d'attaque cryptographique s'agit-il ?
4. Quelle solution proposeriez-vous ?

Exercice 5 : Analyse d'un exploit *race condition* (/4)

On s'intéresse à la vulnérabilité Xorg publiée fin 2011 sous l'identifiant CVE-2011-4029. Cette dernière exploite une « condition de concurrence » (en anglais *race condition*) de Xorg qui permet à un utilisateur local de positionner les droits en lecture sur n'importe quel fichier du système. La sortie console suivante montre une trace des appels système exécutés pour créer ce verrou lors du lancement d'un serveur X sur le display « :1 ».

```
# strace X:1
open("/tmp/.tX1-lock", O_WRONLY|O_CREAT|O_EXCL, 0644) = 0
write(0, "     20093\n", 11)            = 11
chmod("/tmp/.tX1-lock", 0444)           = 0
close(0)                                = 0
link("/tmp/.tX1-lock", "/tmp/.X1-lock") = 0
unlink("/tmp/.tX1-lock")                = 0
```

La trace est composée des étapes suivantes :

1. Ouverture d'un fichier verrou temporaire (/tmp/.tX1-lock)
2. Écriture de l'identifiant du processus (PID) dans ce nouveau fichier
3. Positionnement des droits en lecture pour tous les utilisateurs
4. Fermeture du fichier temporaire
5. Création d'un lien physique avec le véritable nom du fichier verrou (/tmp/.X1-lock)
6. Suppression du fichier verrou temporaire (/tmp/.tX1-lock)

C'est dans la manière dont est manipulé le fichier verrou temporaire (étapes 1 à 4), lors du lancement d'un serveur X, que la faille réside. Voici un extrait de la fonction `LockServer()` du fichier `os/utils.c` :

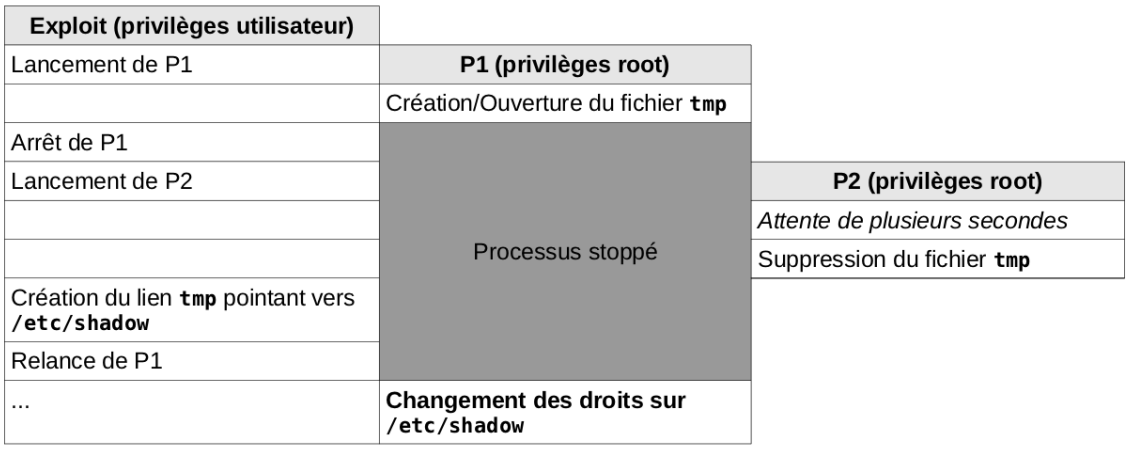
```
294   do {
295     i++;
296     lfd = open(tmp, O_CREAT | O_EXCL | O_WRONLY, 0644);
297     if (lfd < 0)
298        sleep(2);
299     else
300        break;
301   } while (i < 3);
...
314    if (lfd < 0)
315      FatalError("Could not create lock file in %s\n", tmp);
316    (void) sprintf(pid_str, "%10ld\n", (long)getpid());
317    (void) write(lfd, pid_str, 11);
318    (void) chmod(tmp, 0444);
319    (void) close(lfd);
...
328     haslock = (link(tmp,LockFile) == 0);
329     if (haslock) {
...
333       break;
334     }
335     else {
336       /*
337        * Read the pid from the existing file
338        */
339       lfd = open(LockFile, O_RDONLY);
340       if (lfd < 0) {
341         unlink(tmp);
342         FatalError("Can't read lock file %s\n", LockFile);
343       }
```

La fonction `open()` permet la création du fichier `tmp` : en cas de réussite, `open()` assure qu'aucun fichier du même nom n'était présent sur le disque et renvoie un *descripteur de fichier*. A l'inverse de `fchmod()`, la fonction `chmod()` opère sur un *nom* de fichier et non sur un *descripteur* de fichier.

L'exploit de type « Time-Of-Check-To-Time-Of-Use » de la faille CVE-2011-4029, consiste, entre les lignes lignes 296 et 317, à supprimer puis remplacer le fichier *tmp* par un lien symbolique pointant vers un fichier arbitraire, afin que le changement de droits de la ligne 318 soit effectué sur le fichier de notre choix.

La difficulté de l'exploit est de réussir à intercaler précisément ces actions dans l'exécution de `LockServer()`. Le principe général de l'exploit, résumé par la figure 1, consiste notamment :

- à exécuter une première instance (P1) de Xorg contrôlée via les signaux `SIGSTOP` et `SIGCONT` qui permettent respectivement de mettre en pause un processus et de le relancer là ou il s'était arrêté. L'arrêt de la première instance d'Xorg doit se faire immédiatement après la création du fichier temporaire (ligne 296) ;
- à exécuter une deuxième instance (P2) qui supprimera le fichier `tmp` pendant que (P1) est stoppé ;
- à scruter efficacement le système de fichier. Pour cela, l'API `Inotify` permet de reporter à une application tout changement sur un système de fichier au lieu de guetter l'apparition d'un fichier en testant sa présence avec une boucle `while()`.



(Figure 1) Fonctionnement de l'exploit

1. Expliquer le problème de la fonction `LockServer()`.
2. Comment et pourquoi (P2) supprimera le lien physique (que l'on assimilera à un fichier) vers `LockFile` ?
3. Quel est l'intérêt de `Inotify` ? Qu'adviendrait-il si on utilisait une technique naïve de boucle `while()` pour scruter l'activité dans `/tmp` ?
4. Pourquoi le lancement de l'instance (P1) dans l'exploit se fait avec la plus basse priorité ?

5. Quelle correction pourrait-on apporter pour supprimer cette faille ?
6. Dans le *proof of concept* de l'exploit, le fichier sur lequel est positionné `0444` est `/etc/shadow`. Donner des exemples d'impacts que l'exploitation de cette technique permettrait d'avoir. Quel autres fichiers pourrait-on vouloir utiliser ?

Cet exploit sera détaillé dans la section exploit corner *du numéro 60 de la revue Misc. Merci à Julien Lantheaume pour l'utilisation d'une partie de son article pour cet examen.*

1 Corrections

Solution de l'exercice 1

1. le risque est le produit d'une gravité (dommage) et d'une vraisemblance (probabilité). Pour traiter le risque, on peut :
   – l'éviter (changer de contexte que la question ne se pose plus),
   – le réduire (en diminuant la gravité ou la vraisemblance),
   – le prendre (on ne change rien)
   – le transférer (on externalise/partage le risque, via une assurance par exemple)
2. symétrique : rapide mais nécessite un canal sûr pour transmettre la clef. Asymétrique : les clefs sont publiques (pas de canal sûr nécessaire mais reste le problème de l'authentification), on peut signer, le calcul est cher.
3. ce sont trois façons de s'attaquer à des passwords hashés : soit par test exhaustif des possibilités (*brute force*), d'après une liste connue de mots (dictionnaire), ou avec une liste pré-calculée de clairs hashés (stockées en chaînes pour diminuer l'espace requis, *rainbow tables*).
4. moindre privilège : on ne donne que les droits *nécessaires* aux utilisateurs, pas des droits suffisants. Séparation des tâches : on découpe les activités en imposant que plusieurs utilisateurs interviennent dans leurs réalisations.
5. le modèle RBAC c'est un modèle structuré (agnostique DAC/MAC) qui vise à rapprocher la gestion des droits des organisations pour la faciliter, en factorisant les opérations administratives selon des rôles possiblement hiérarchisés. Des variantes permettent l'expression de contraintes pour pouvoir exprimer des politiques de séparation des tâches.
6. c'est une façon d'exploiter des erreurs de programmation où des paramètres utilisateurs écrits dans la pile écrasent les adresses de retour des fonctions, ce qui peut permettre de diriger le flot d'exécution à du code arbitraire (*shellcode*).

Solution de l'exercice 2

1. On note $X_\nabla = X \cup \{\nabla\}$ où ∇ est un élément qui n'appartient pas à X. Les principaux ensembles sont les événements Ev, les utilisateurs U, les formulaires F, rôles R et les épisodes Ep. On suppose l'existence des relations $\mathcal{URA} \subseteq U \times R$ et $Pra \subseteq R \times F$ entre utilisateur et formulaires. On se dote des fonctions suivantes :
   – $form : Ev \to F$
   – $author : Ev \to U$
   – $episode : Ev \to Ep_\nabla$
2. La définition des classes est assez brouillone. On remarque qu'il n'y a que trois possibilités parmi les quatre théoriquement possibles voir/être vu. La manquante est celle des utilisateurs qui produisent des événements visibles mais qui ne peuvent pas voir les événements des autres.
3. On associe à chaque épisode un sous-ensemble des utilisateurs, un pour chaque classe. On note un premier S si l'utilisateur peut voir les événements de l'épisode (X s'il ne peut pas) et un second S si les événements qu'ils produit sont visibles (X s'il ne peut pas) :
   – $SS : Ep \to \wp(U)$ pour *full*
   – $SX : Ep \to \wp(U)$ pour *restricted*
   – $XS : Ep \to \wp(U)$ pour la classe manquante
   – $XX : Ep \to \wp(U)$ pour *exclusive*
   Il est censé que les classes d'utilisateurs forment une partition d'un sous-ensemble de U, certains utilisateurs pouvant ne faire partie d'aucune classe, soit $SS(e) \cap SX(e) = \emptyset$, $SS(e) \cap XS(e) = \emptyset$ etc. pour chaque épisode.
4. On construit $RBAC = \{(u, f) \mid \exists r.(u, r) \in \mathcal{URA} \wedge (r, f) \in Pra\}$

– $lecteur : Ep \to \wp(U), lecteur(e) = SS(e) \cup SX(e)$
– $invisible : Ep \to \wp(U), invisible(e) = XS(e) \cup XX(e)$

5. On définit la fonction auxilliaire $aux : Ep \times U \times U \to \{tt, ff\}$ par $aux(e,u,a) = (a = u) \vee (u \in lecteur(e) \wedge a \notin invisible(e))$. On peut alors définir $auth : Ev \times U \to \{tt, ff\}$ par $auth(u,e) = (episode(e) = \nabla) \vee aux(episode(e), u, author(e))$
6. On définit l'accès RBAC à partir des formulaires $authRBAC : U \times Ev \to \{tt, ff\}$ par $authRBAC(u,e) = (form(e) = f) \wedge (u,f) \in RBAC$ puis on les combine $access(u,e) = authRBAC(u,e) \wedge auth(u,e)$
7. – Episodes
   – $XX(E_1) = \{Guru\}, SS(E_1) = \{MyPhys, MyNurse\}$
   – $SX(E_2) = \{MyPhys, OtherPhy\}, SS(E_2) = \{MyNurse\}$
   – Matrice
   – $(Guru, Physician) \in URA, (MyPhys, Physician) \in URA, (OtherPhy, Physician) \in URA, (MyNurse, Nurse) \in URA$
   – $(Physician, General) \in PRA, (Physician, Treatment) \in PRA, (Nurse, General) \in PRA$
   – Dossier médical
   – $e_1 = (General, MyNurse, \nabla)$
   – $e_2 = (Treatment, MyPhys, \nabla)$
   – $e_3 = (General, MyPhys, E_1)$
   – $e_4 = (Treatment, Guru, E_1)$
   – $e_5 = (Treatment, MyPhys, E_2)$
   – $e_6 = (General, MyPhys, E_2)$
   – $e_7 = (General, OtherPhy, E_2)$

	∇ e_1	∇ e_2	E_1 e_3	E_1 e_4	E_2 e_5	E_2 e_6	E_2 e_7
MyNurse	*tt*	*ff*	*tt*	*ff*	*ff*	*ff*	*ff*
MyPhysician	*tt*	*tt*	*tt*	*ff*	*tt*	*tt*	*ff*
Guru	*tt*	*tt*	*ff*	*tt*	*ff*	*ff*	*ff*
OtherPhysician	*tt*	*tt*	*ff*	*ff*	*ff*	*ff*	*tt*

Solution de l'exercice 3

1. On va effectuer une injection SQL car les chaînes de l'utilisateur ne sont pas saines. On va avoir ainsi une condition *where* toujours vraie : on va avoir la liste complète des items, sans filtrage sur l'identifiant du propriétaire.
2. plusieurs cas possibles :
   – les requêtes multiples ne sont pas autorisées, il ne se passera rien ;
   – les requêtes multiples sont autorisées, si l'utilisateur du tiers a le droit de suppression alors le contenu de item sera supprimé, sinon, il ne se passera rien.
3. en général, la perte de confidentialité et d'intégrité des données du SGBD, ce qui peut se traduire concrètement de différentes façons selon les applis. Par exemple, s'il s'agit d'une interface de gestion de compte ou d'authentification on va peut-être pouvoir s'identifier sans compte, éventuellement comme administrateur, accéder à la liste des utilisateurs et de leurs passes hashés, créer/supprimer des utilisateurs...
4. L'idée générale est que dans une chaîne envoyée au SGBD, il faut séparer la partie commande des données saisies par l'utilisateur. Pour cela on peut :
   – nettoyer les saisies de l'utilisateur en dé-spécialisant les caractères spéciaux (côté tiers ou SGBD) ;
   – utiliser des procédures stockées dans le SGBD ;
   – utiliser des requêtes préparée (*prepare statement*).

Solution de l'exercice 4

1. On est dans *Porous Defense* : le mécanisme choisi ici est inadapté, plus particulièrement *Use of a Broken or Risky Cryptographic Algorithm*. On peut aussi dire que c'est une augmentation locale de privilège (*local privilege escalation*).
2. c'est l'exploitation d'une faille jusqu'ici inconnue (*0-days*) utilisé par le vers Stuxnet pour se propager, vers de haute technologie ciblant les infrastructures industrielles, parmi les 4 qu'il comptait.
3. c'est une attaque à la seconde pré-image (pour une fonction de hashage f, connaissant $h = f(m)$, trouver $m \neq m'$ tel que $f(m) = f(m')$). C'est possible car CRC32 n'est pas un hash cryptographique : il n'est pas fait pour résister à des attaques (et l'espace de recherche est trop petit). Pour construire m', on peut s'appuyer sur m modifié et remplir les commentaire xml avec les bons caractères pour faire coïncider les hashés.
4. utiliser un vrai hash cryptographique comme MD5 ou SHA.

Solution de l'exercice 5

1. `chmod()` opère sur un *nom* et `open()` sur un *descripteur* : rien ne garantit donc que le fichier dont les droits modifiés soit celui ouvert.
2. si on entre dans un état ou le fichier pointé est invalide, alors la ligne 341 supprimera le fichier. On pourra alors en mettre un autre à la place : celui sur lequel P1 effectuera `chmod()`.
3. le temps est très serré pour ces exploits : un polling brutal serait inefficace car pas assez précis, le taux de réussite de l'exploit serait quasi nul.
4. idem que précédent, mais avec moindre importance.
5. la correction la plus simple est d'utiliser `fchmod()`. De manière générale, on peut aussi repenser l'utilisation des locks dans `/tmp`.
6. avec `/etc/shadow` on accède aux hashés des mots de passes des utilisateurs, pour pouvoir ensuite les attaquer (cf. question 3 de l'exercice 1). Avec cet exploit, on peut en fait accéder *en lecture* à n'importe quel fichier du système, y compris des partitions entières que l'on pourra dumper, des certificats, des clefs, des fichiers de configurations etc.