Synthèse
Sécurité Applicative

Troisième Bloc
Sécurité des systèmes
Année académique 2021-2022

Rédigé par Sénéchal Julien

30 Décembre 2021

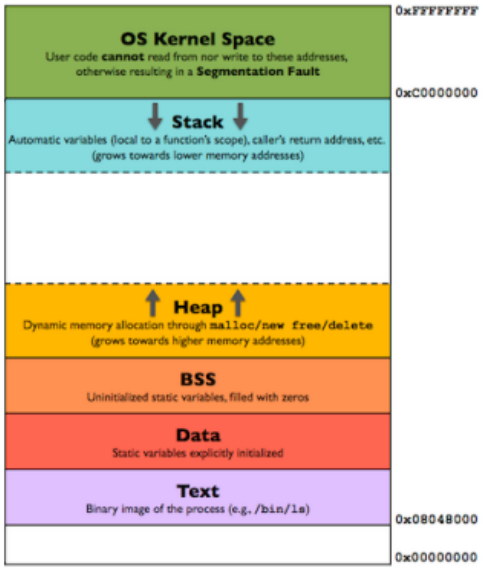
Table des matières

1 Architecture x86_64

Registres généraux d'une architecture 64 bits :

- rax, rbx, rcx, rdx : extended registers
- rbp, rsp : base/stack pointer
  rbp indique le haut de la stack et rsp le bas de la stack. Lors d'une entrée dans une nouvelle fonction, rbp est push sur rsp. Lorsqu'une nouvelle variable est créée, rsp descend (et donc la stack également). Une adresse de retour se trouve sur rbp.
- rsi, rdi : source/destination index

En 64 bits, si programme très simple : Pas besoin de RAM car beaucoup de registres



- text : les instructions du programme en format binaire. Exécutable, read-only.
- data : les variables globales et statiques initialisées. Read-only.
- bss : les variables globales et statiques non-initialisées.
- heap : la mémoire dynamique, alloué avec malloc() par exemple.
- stack : les variables locales, les paramètres et l'adresse de retour de fonction.

Assembleur = Langage intermédiaire entre le binaire et le code source

1.1 Vulnérabilités

1.1.1 Dénifitions

- Deadlock :
  Lorsque chacun des programmes n'est plus en mesure d'effectuer une action attendant tous une ressource. (Cause : obtention simultanée et accès exclusif)

- Race condition :
  Lorsque 2 thread/processus accèdent à une ressource en même temps, étant donné qu'ils auront tous les 2 lu les données avant l'écriture, celle-ci vont procéder au même calcul et donc donner un résultat erroné. Exemple : A et B lisent un entier, tous 2 retiennent 17 et l'incremente de 1. La valeur écrite sera donc 18 au lieu de 19 si les 2 threads avaient s'étaient suivi.
  Solutions : Utiliser des locks (accès exclusif).

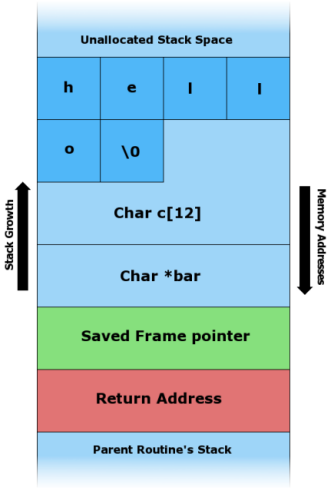
- Fork bomb :
  Boucle de fork amenant au remplissage de la table des processus ainsi que toute la mémoire disponible.

- Memory Leak :
  Pas de libération de la mémoire louée dynamiquement dans la heap (par un malloc par exemple).

- Dangling pointers :
  Mémoire de pointeur libéré mais toujours disponible. Peut aussi être due en utilisant l'adresse d'une variable n'existant que dans une fonction.

- Segmentation fault :
  Déréférencement de NULL ou buffer overflow. (Lecture/Ecriture dans de la mémoire protégée)

- Stack buffer overflow :
  Ecrit plus de donnée que possible. Si il s'agit d'une variable locale = stack. Trop de donnée = débordement sur la frame actuelle, peut-être sur la frame précédente, l'adresse de retour et encore la frame précedente. Si l'on veut comparer la figure suivante avec le schéma de la mémoire présenté dans les premières pages, il nous faut alors mettre celle-ci à l'envers pour être dans le même sens.



- Time/Event bomb :
  Action malveillante se déclenchant à un évènement ou après un certain temps.

- Code injection :
  Le code exécute l'entrée utilisateur.

- Zip bomb :
  2 types :
  — recursif (peut-être bloqué par un unpacker ou même un anti-virus)
  — non-recursif (beaucoup plus subtil)

  Va saturer le système lors de la décompression (mémoire, puissance de calcul).

- XML bomb :
  Chaque entité du XML peut faire référence à toutes les autres entités et ce un certain nombre de fois. (Example : Entité 9 appelle les 8 autres et la 8ème appelle les 7 autres et etc...)

- YAML bomb : Pareil que la XML bomb.

1.1.2 Race condition : Time of check/Time of use

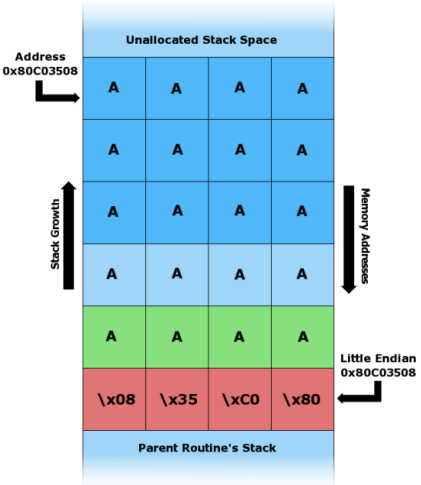
1. Le fichier à modifier possède des droits SUID
2. Temps de pause entre lecture des droits (time of check) et écriture dans le fichier (time of use). Peut-être causé par l'attente d'une entrée utilisateur ou un simple sleep().
3. Possibilité de remplacer, durant ce délai, le fichier par un symlink vers un fichier que l'on veut modifier

1.1.3 Stack smashing

Peut survenir si :

1. Exploitation d'un stack buffer overflow
2. L'attaquant connaît la taille du buffer à remplir
3. Il peut construire les données à copier de façon à remplacer l'adresse de retour par une adresse de son choix

Il peut alors simplement insérer du code exécutable, et modifier l'adresse de retour par l'adresse vers le code injecté. Il peut ainsi injecter du code ayant les mêmes droits que ceux du programme.



Solutions :

- Stack non exécutable (-z execstack)
- Canaris : flag avant l'adresse de retour (-fno-stack-protector)
- ASLR : emplacement random dans la mémoire à chaque execution (echo '0' > /proc/sys/kernel/randomize_va_space)

Peuvent être contournées mais difficilement.

1.1.4 Format string vulnerability

Danger : utilisation de printf() en mettant directement la variable sans utiliser de formatage.

Possibilité donc de :

↪ Afficher la stack :

```
$ ./a.out $(python -c 'print "%08x" * 32')
```

Format%x affiche la valeur en hexa. Si rien n'est fourni à printf comme valeur, la fonction va récupérer les valeurs de la stack.

↪ Crash :

```
$ ./a.out $(python -c 'print "%s" * 100')
```

Format %s est utilisé pour afficher une chaine de caractère, à partir d'un pointeur. Sans valeurs, printf va utiliser les valeurs de la stack comme pointeurs, ce qui mène rapidement à une segfault.

↪ Accès à la mémoire arbitraire :

```
$ ./a.out $(python -c 'print <address> + "%x" * n + "%s"')
```

Comme notre argument de programme se retrouve sur la stack, nous pouvons y écrire une adresse. Et comme nous disposons de %s, qui affiche le contenu présent à une adresse, il est possible d'afficher la mémoire de toute adresse donnée. Il s'agit simplement de trouver le bon nombre de bytes à retirer de la stack avant notre adresse.

↪ Écriture dans la mémoire :

```
$ ./a.out $(python -c 'print <address> + "%x" * n + "%n"')
```

L'idée est identique à l'accès à la mémoire mais le %n va écrire le nombre de caractères affichés précédemment (entier) à l'adresse du paramètre fourni, autrement dit l'adresse que nous avons écrite sur la stack.

2 Analyse

2.1 Statique

2.1.1 Basique

Analyse dans accéder au code source :

- strings (Extraction des strings)
- file (Analyse du fichier binaire)
- Vérifier les permissions
- Metadata (exiftool)
- Antivirus (virustotal, ClamAV, etc.) :
  Conseillé d'en utiliser plusieurs, ce que fait virustotal. Il existe 2 stratégies :
  1. Statistique : Hash comparé à une base virale.
  2. Heuristique : Vérifier le comportement du programme pour trouver des opérations suspectes (possibilité de faux-positif).

2.1.2 Avancée

Analyse sans exécution du code source :

- Ouverture du code source (si possible)
- Ghidra (Décompilateur : fournit le code source)
- GDB (Désassembleur : fournit le langage d'assemblage)
- IDA (Désassembleur : fournit le langage d'assemblage)
- Decompile++ (Pour Python bytecode)
- Analyseur de code (Pylint, RATS, etc.)
- A la main :
  - ↪ Les librairies et fonctions utilisées
    - eval, exec, etc...
    - strcpy (écriture en mémoire)
    - printf, scanf
    - sys, os, etc.
  - ↪ Les entrées/sorties
    - provenance (utilisateur, fichier, socket)
    - validation
    - droits de lecture ou écriture
    - destination
    - infos sensibles en clair
  - ↪ Interactions avec le système
    - commandes ou executables externes
    - réseaux (socket, ftp, etc.)
    - périphériques
    - lecture, ecriture, copie, etc.
    - variables d'environnement

2.2 Dynamique

2.2.1 Basique

Analyse durant l'exécution du programme, dans un environnement contrôlé (sandbox) :

- top
- ps aux
- pmap
- pidstat
- /proc/<PID>/status
- /proc/<PID>/FD (permet de voir des fichiers en cours de lecture/modification/création)3
- lsof (affiche tous les fichiers utilisés sur le système par tous les process)
- printenv (regshot sur Win) permet d'afficher les variables d'environnement. Il peut être intéressant de faire hash avant et après pour voir si il y a eu un changement
- strace
- autres outils d'analyse des processus
- Fuzzing (ex : boofuzz, radamsa)
- Réseaux : Wireshark, netstat, ss, fuser

2.2.2 Avancée

Analyse durant l'exécution du programme, grâce à un debugger pour inspecter la mémoire (ex : GDB).

2.3 Contre-mesure à l'analyse

- Anti-disassembly
  Introduction de code difficile à comprendre (ex : double jump conditionnel) voire incorrect.
- Anti-debug :
  Modifier le comportement d'un programme si il passe dans un debugger. L'execution dans un debugger est différente et même parfois détectable grâce à des API de l'OS (windows).
- Anti-VM :
  L'execution dans une VM est facilement détectable et peu donc modifier son comportement pour paraître innofensif.
- Obfuscation :
  Rendre le code insupportable à lire pour brouiller les pistes (ex : brainfuck)
- Unpacking :
  Programme se présentant sous forme d'un package executable. Évite l'analyse du code exécutable malveillant.

3 Prévention

La conception d'un programme doit être basé sur la sécurité. Importance de la *Security by design* :

- Hardening :
  Limiter les surfaces d'attaque : HTTP to HTTPS, limiter les entrées (barre de recherches par ex), diviser les services plutôt qu'une seule grosse application.
- Least Privilege :
  Pas de shell root, pas de connexion avec un compte admin sur les services mais un compte juste suffisant (ex : db), pas d'accès en lecture sur tout le FS
- Security by default :
  Si le niveau de sécurité peut être modifié, le niveau de sécurité doit être le plus élevé par défaut. (utilisation de rôle au lieud d'un compte admin, https, port aléatoire, etc.)
- Defense in depth :
  Ne pas se contenter d'une seule vérification de sécurité. Ex : Même si il s'agit d'un intranet, il faut tout de même s'identifier pour accéder au réseau. (2FA, Vérification systématique des droits, etc.)
- Failing securely :
  Si jamais une erreur doit survenir pendant le dérouleement d'une vérification de droits, la fonction doit refuser les droits par défaut et ne pas laisser le rôle "à déterminer" (ex : try return isAdmin() ; except return False ;).
- Avoid external trust :
  Méfiance envers les services externes à notre contrôle (API, ISP, CDN, etc.)
- Seperation of duty :
  Utilisation typique :
  ↪ Une entité contrôle si l'action peut être effectuée (ex : droit, ACL, etc.)
  ↪ Une entité fait l'action (ex : user)
  ↪ Une entité note l'action effectuée (ex : logging)
  De cette manière on évite qu'une entité fasse quelque chose et le cache de lui même.
- KISS = **K**eep **I**t **S**imple **S**tupid :
  Le système doit être le plus simple possible. Plus il est complexe, plus il sera difficile de le comprendre et ne le sécuriser.
- Avoid security by obscurity
  Ne jamais croire que rendre la lecture d'un programme difficile le rend sécurisé. But ? Décourager !
- Open Security (Principe de Kerchoff) Dans tout programme, partir du principe que l'attaquant en aura la même connaissance que nous. Seul la clé doit rester secrète. Ce principe dirige naturellement vers la mise à disposition de tous du code source (open-source). Cela peut avoir un impact important sur la sécurité :
  ↪ S'assurer qu'il a bien été conçu et que la divulgation ne constitue pas un risque
  ↪ Les failles de sécurité s'y trouvant ont plus de chances d'être découvertes et corrigées
- Safe language :
  - Adapté
  - Memory safe (pas du C kappa)
  - Typage sûr (en général, un typage fort et statique est considéré comme plus sûr. Par exemple le JavaScript est tout sauf un language au typage sûr)
- Defensive programming :
  Tout ce qui peut mal se passer doit être pris en compte :
  - Inputs (valider le format et faire de la sanitization)
  - Erreur (Gérer les erreurs grâce à des exceptions)
  - Cas exceptionnels (Liste vide, division par 0, tri avec doublons, etc.)
  - Passer par des batteries de tests. Son importance à amené au TDD (Test driven development) : Avant d'écrire du code, les tests sont créé pour vérifier les spécifications de la méthode ou fonction.
  - etc.

4 Défense

4.1 Stratégie de protection

Règle numéro 1 : Tout logiciel doit être traité comme une menace potentielle ! C'est pour cela que tout logiciel doit avoir accès aux ressources juste nécessaire, pas plus pas moins.

Bonnes pratiques et stratégies de protection :

- Pour chaque logiciel, créer un utilisateur et un groupe.
- Limiter l'accès de cet utilisateur au FS.
- Si le logiciel est lancé par un autre utilisateur, passer sur son utilisateur attribué dès que possible (setuid() & setgid()).
- Si besoin de droits admin, s'assurer que les droits soient relâché dès que possible !
- Limiter les ressources matérielles (/etc/security/limits.conf, ulimit & quotatool). Le but est d'empêcher le DOS.
- Bonne gestion du réseaux en mettant par exemple :
  - la création d'alerte lors d'une communication suspecte
  - les iptables pour restreindre certain utilisateurs/processus (module *owner*)
  - le monitoring des requêtes DNS
- Le sandboxing :
  - Jailing avec Chroot :
    Fournir une version restreinte du système en modifiant la racine du système de fichiers. Possibilité de modifier également le PATH enlever des exécutables. Idéal pour un hébergement mutualisé/virtualisé (plusieurs sites sur le même serveur).
  - Machine virtuelle :
    Simulation d'un système complet. Une faille dans l'hyperviseur pourrait permettre à un malware de sortir. De plus, celui-ci peut adopter un comportement différent quand il est dans une VM.
  - Containers :
    Entre le jailing et la VM. Basés sur les cgroups qui attribuent des restrictions à un groupe de processus, ils permettent l'exécution de processus complètement isolés des autres mais utilisant les capacités du même système hôte.
- Sauvegarde de données :
  - Facile à metre en place et à restaurer
  - Double copie (locale et distante)
  - Fréquence suffisante
  - Conservation dans le temps suffisante
- Sauvegarde du système :
  - Système bare metal : image des partitions systèmes
  - Système virtuel : snapshots ou copie des disques
- Mettre en place des solutions de remote logging

4.2 Contre quoi se défendre ?

- Tout applicatif ou logiciel qui permet une violation du CIA (Confidentiality/Integrity/Availability).
- Tous les logiciels posent des menaces plus ou moins importantes

4.3 Menaces

- Lecture de données
- Modification des données
- Espionnage de l'activité
- Détournement de ressources matérielles (ex : minage)
- Déni de service
- Spoofing
- etc.

4.4 Définition d'un logiciel menaçant

Une telle définition n'existe pas car cela dépend du contexte et du cas d'utilisation.

4.5 Detection par logiciel

La détection de logiciel malveillant de manière parfaite est impossible à l'aide d'une machine

5 Pratique

5.1 GDB

Commandes utiles :

- disassemble <function>
- run
- break <fonction/ligne> : ajout de breakpoint
- continue : continue après le break
- print &<x> : affiche m'adresse d'une variable/fonction
- print *<x> : affiche la valeur d'une variable/fonction
- x/<n> &<y> : affiche n bytes après l'adresse de y
- x/<n><o> $rsp : afficher n bytes de la stack sous forme de o :
  - d : signé
  - x : hexa
  - u : non-signé

5.2 Comment faire un stack smashing

Partons du principe qu'on le sait déjà vulnérable :

1. Calculer le buffer :
   - GDB
     - Ouvrir le binaire avec gdb
     - Mettre un break après le strcpy et run du padding en argument
     - print $rsp
     - print $rbp
     - Grâce à *x/100xg $rsp*, repérer notre buffer et compter le nombre de byte restant jusqu'à arriver à l'adresse de $rbp.
     - Noter également l'adresse du debut de notre buffer
     - La taille du buffer = nombre de caractère mis en argument + le nombre compté
   - La taille du buffer peut être plus facilement trouvée grâce a Ghidra
2. Taille du padding = Taille du buffer + taille RBP (8) - taille du code
3. Commande finale :

   > ./binary $(python -c '<code> + "A" * <Taille padding> + <adresse du buffer>[: :-1]')

   L'adresse du buffer doit s'écrire de manière à ce que 0x7fffffffdee0 devienne : "\x7f\xff\xff\xff\xde\xe0". Le little-endian se fait grâce à [: :-1]

Il est important de savoir qu'à cause des variables d'environnements différentes entre gdb et votre shell, l'adresse du buffer n'est pas la même. Voilà pourquoi toute cette explication ne sert à rien.

5.3 Exemple de tests en python

```python
class TestMethods(unittest.TestCase):

    def testsum(self):
        calculator = Calculator()
        self.assertEqual(calculator.sum(5,6.5), 11.5)
        self.assertEqual(calculator.sum(5,-2), 3)
        self.assertEqual(calculator.sum(5, 2), 7)
        self.assertEqual(calculator.sum(5.5,6.5), 12)
        self.assertEqual(calculator.sum(5,0), 5)
        self.assertEqual(calculator.sum(-5,0), -5)
        self.assertEqual(calculator.sum(2.5,0), 2.5)
        self.assertEqual(calculator.sum(5,-5.5), -0.5)

    def testdiv(self):
        calculator = Calculator()
        self.assertEqual(calculator.div(15.5,0.5), 31)
        self.assertEqual(calculator.div(15,5), 3)
        self.assertEqual(calculator.div(-15,5), -3)
        self.assertEqual(calculator.div(15,-0.5), -30)
        with self.assertRaises(ZeroDivisionError): calculator.div(15.5,0)
```

Lancement des tests avec : unittest.main()

5.4 Comment faire du jailing

1. Créer le dossier du chroot et mettre root comme owner :

   > # mkdir /home/jail
   > # chown root :root /home/jail

2. Lister et copier tous les fichiers nécessaires au lancement d'un shell et des commandes voulues dans le dossier précédemment créé :

   > # ldd /bin/{bash, ls, cp, rm}
   > # mkdir /home/jail/{bin,lib64,lib}
   > # cp /bin/bash /home/jail/bin
   > # cp /lib/lib/x86_64-linux-gnu/libtinfo.so.5 /home/jail/lib/
   > etc.

3. Créer le/les utilisateur(s) et/ou le groupe

   > # groupadd jail (facultatif)
   > # useradd myuser
   > # passwd myuser
   > # usermod -aG jail myuser (facultatif)

4. Ajouter les lignes suivantes à la fin du fichier /etc/ssh/sshd_config (mettre User ou group dépendant de votre cas) :

   > Match User(/group) myuser(/jail)
   > .....ChrootDirectory /home/jail

5. Vérifier que le shell par défaut de/des utilisateur(s) soit bien celui ajouté au dossier /home/jail

5.5 Retrouver les arguments dans la mémoire - GDB

1. *break main*
2. *info frame* :

```
0x7fffffffe398: "/home/kali/Desktop/Labo6/a.out"
(gdb) info frame
Stack level 0, frame at 0x7fffffffdf60:
 rip = 0x55555555515d in main (autonomie.c:6); saved rip = 0x7ffff7e17e4a
 source language c.
 Arglist at 0x7fffffffdf50, args: argc=3, argv=0x7fffffffe048
 Locals at 0x7fffffffdf50, Previous frame's sp is 0x7fffffffdf60
 Saved registers:
  rbp at 0x7fffffffdf50, rip at 0x7fffffffdf58
(gdb)
```

3. On lit les données à ces emplacements mémoire. J'ai personnellement eu des adresses à cette endroit dans la mémoire :

```
(gdb) x/3xg 0x7fffffffe048
0x7fffffffe048: 0x00007fffffffe398       0x00007fffffffe3b7
0x7fffffffe058: 0x00007fffffffe3bc
(gdb)
```

4. Il ne suffit plus que de lire le contenu de ces adresses :

```
(gdb) x/s 0x00007fffffffe398
0x7fffffffe398: "/home/kali/Desktop/Labo6/a.out"
(gdb) x/s 0x00007fffffffe3b7
0x7fffffffe3b7: "yolo"
(gdb) x/s 0x00007fffffffe3bc
0x7fffffffe3bc: "message"
```

5.6 Attaque par format string

1. Code vulnérable :

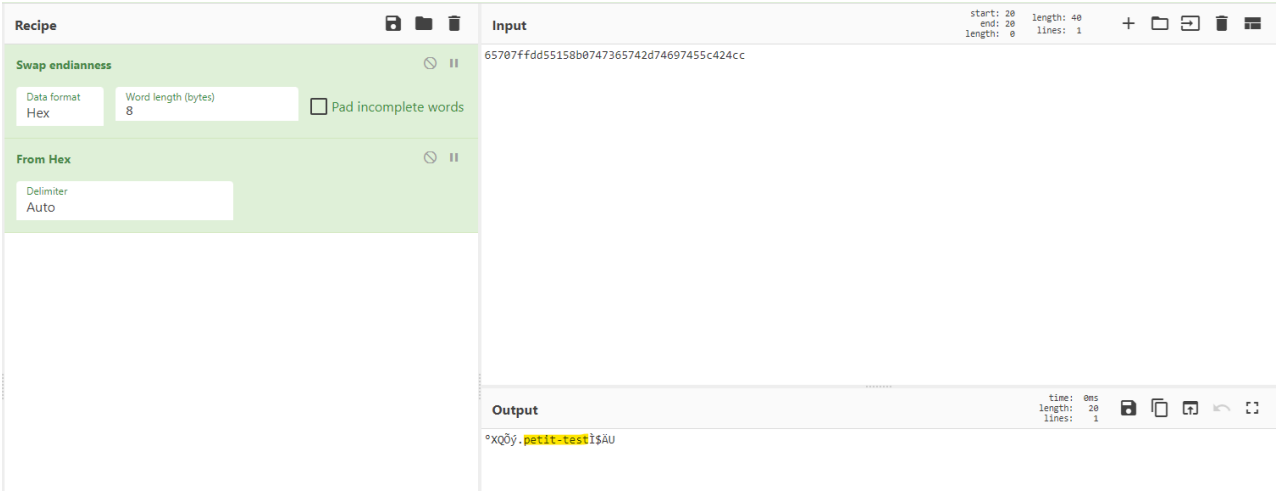
```c
#include <stdio.h>
#include <string.h>

void main(int argc, char* argv[]){
    char buffer[10];
    strcpy(buffer, argv[1]);
    printf(argv[2]);
}
```

2. Afficher la stack et lire l'hexadecimal :

   $ ./a.out "petit-test" "`python -c "print '%lx' * 10 "`"

```
┌──(kali㉿kali)-[~/Desktop/Labo6]
└─$ ./a.out "petit-test" "`python -c "print '%lx' * 10 "`"
7ffdd55163ca7ffdd55157b67ffdd55163ca07fd47fe392f07ffdd55158b8324cc006065707ffdd55158b0747365742d74697455c424cc0100
```



5.7 Créer un daemon DNS Monitoring

1. Créer un groupe ayant les droits de lancer tcpdump ainsi qu'un utilisateur dans ce groupe :

   # groupadd pcap

   # usermod -aG pcap <user>

   # chgrp pcap /usr/bin/tcpdump

   # setcap cap_net_raw,cap_net_admin=eip /usr/bin/tcpdump

   # ln -s /usr/bin/tcpdump /usr/local/bin/tcpdump

2. Créer un service dans */lib/systemd/system/<nom>.service* et y mettre une configuration comme celle-ceci :

   [Unit]
   After=network.target

   [Service]
   ExecStart=/usr/bin/tcpdump -i eth0 ip and 'port 53' -w /home/mypcap/output.pcap
   Type=simple

   [Install]
   WantedBy=multi-user.target

5.8 Auditd

1. Ajouter auditd au grub
   (a) Ouvrir /etc/default/grub
   (b) Modifier la ligne GRUB_CMDLINE_LINUX_DEFAULT en y ajoutant "audit=1"
   (c) Update le grub avec :

   # grub-mkconfig -o /boot/grub/grub.cfg

2. systemctl enable auditd
3. Exemple de règles temporaires :
   — auditctl -w /etc/passwd -p rwa
   — auditctl -a always,exit -S all -F uid=1002
   — auditctl -a always,exit -F arch=b64 -S chmod,open,kill -k test
   (-k sert à lier le log avec une clé afin de le retrouver plus facilement)
4. Pour créer des règles permanentes, il suffit d'ajouter les options dans le fichier /etc/audit/rules.d/audit.rules et de redémarrer le service.
5. Lire le résultat de l'audit dans /var/log/audit/audit.log

5.9 Limiter l'espace disque d'un utilisateur avec un disque virtuel

1. Créer un dossier dans /media/ :

   ```
   mkdir /media/test
   ```

2. Copier et convertir /dev/zero en une image avec les quotas requis :

   ```
   dd if=/dev/zero of=/media/test/client.img bs=512k count=200
   ```

3. Formater ce disque :

   ```
   mkfs.ext4 /media/test/client.img
   ```

4. Créer le répertoire et monter le disque :

   ```
   mkdir /home/quotatest
   mount -o loop /media/test/client.img /home/quotatest
   ```

5. Pour rendre ce montage permanent, modifier /etc/fstab en rajoutant ce qui suit :

   ```
   /media/test/client.img /home/client ext4 loop 0 2
   ```

6. Créer l'utilisateur et le nommer propriétaire de son home :

   ```
   adduser quotatest
   chown -R quotatest :quotatest /home/quotatest
   ```