ChatGPT

Cours approfondi sur les débordements de tampon et les protections des exécutables

Lorsque l'on compile un programme C/C++ sous Linux, le résultat est un fichier **ELF** (Executable and Linkable Format). Ce format contient des métadonnées décrivant l'architecture (32 bits vs 64 bits), le type de liaison (dynamique vs statique), le chemin de l'interpréteur, etc. Par exemple, la commande `file` appliquée à un exécutable 64 bits affiche quelque chose comme :

```
$ file hello
hello: ELF 64-bit LSB executable, x86-64, … dynamically linked, … not stripped
```

Cet exemple indique que l'exécutable est au format 64 bits (**ELF 64-bit LSB**), lié dynamiquement (`dynamically linked`) et qu'il *n'est pas dépouillé* de ses informations de débogage [1]. En pratique, `file` est très utile pour connaître l'architecture du binaire (x86 vs ARM, 32 vs 64 bits), le bitness (LSB/MSB), le type de liaison et si les symboles de débogage sont présents ou non.

- **Architecture 32-bit vs 64-bit** : sans option particulière du compilateur, un système x86_64 produira par défaut un ELF 64 bits (x86-64). Avec l'option `-m32` on peut obtenir un binaire 32 bits (x86) [1].
- **Liaison statique vs dynamique** : la **liaison statique** intègre le code des bibliothèques directement dans l'exécutable lors de la compilation, ce qui produit un fichier plus volumineux mais indépendant des bibliothèques externes [2] [3]. La **liaison dynamique** (le plus courant sous Linux) laisse le code des bibliothèques partagées (`.so`) séparé : l'exécutable contient un petit stub de démarrage qui charge ces bibliothèques au lancement. Cela garde les fichiers plus petits et partage les bibliothèques en mémoire [4] [3]. Par exemple, pour un binaire statique on verra dans `file` l'étiquette « statically linked » et la commande `ldd ./program` répondra « not a dynamic executable » [3] [5].
- **Exécutable dépouillé (stripped) vs non dépouillé** : dépouiller (`strip`) un binaire signifie enlever les symboles de débogage et les noms de fonctions, ce qui réduit la taille mais complique l'analyse. Un binaire *non* dépouillé contiendra ses symboles (utile au débogage). Dans notre exemple, `file` mentionne « not stripped » pour signifier que les symboles sont présents [1] [3].

Pour inspecter un binaire plus en profondeur, on peut utiliser `readelf` ou `objdump` (par exemple `readelf -h prog` pour l'en-tête ELF, `readelf -l prog` pour les segments, `objdump -d prog` pour le désassemblage) et `ldd prog` pour lister les bibliothèques dynamiques. L'outil **checksec** affiche directement un tableau des protections activées (RelRO, Canary, NX, PIE, Fortify, etc.) [6].

Protections de sécurité des binaires

Lors de la compilation, on peut activer diverses protections pour limiter les risques d'exploit par débordement de tampon ou autres attaques. Voici les principales :

- **Canary (Stack Canaries)** – C'est un mécanisme de détection de débordement de pile. Le compilateur insère une valeur aléatoire (le canari) juste avant l'adresse de retour dans chaque trame de pile. Avant que la fonction ne retourne, ce canari est vérifié : si sa valeur a changé (par exemple écrasée par un débordement), le programme termine immédiatement [7]. En pratique, on active cela avec l'option GCC `-fstack-protector` ou `-fstack-protector-all`. La présence de canaries se traduit dans `checksec` par « Canary found » ou similaire [7].
- **NX (Non-Executable Stack ou DEP)** – L'option NX (No-eXecute) interdit l'exécution de code dans certaines zones mémoire marquées en écriture (typiquement la pile). Ainsi, même si un attaquant injecte du code machine dans la pile, le processeur empêche son exécution. Cette protection est activée par défaut avec GCC moderne. On peut vérifier via `readelf -l prog | grep GNU_STACK` (la colonne `RWX` ne doit pas comporter le `E`). Sous Linux on peut la désactiver avec `-z execstack`. Le résultat est souvent résumé dans checksec : « NX enabled » si la pile est non exécutable [8].
- **PIE (Position Independent Executable)** – Un exécutable PIE est lié de façon position-indépendante, ce qui permet au chargeur de l'OS de le placer à une adresse différente à chaque exécution (comme pour une bibliothèque partagée). Ainsi, les adresses du code statique sont *randomisées* (ASLR). Si un binaire n'est *pas* PIE, son segment de code sera toujours chargé au même endroit en mémoire, facilitant l'exploitation. En général on active PIE avec `-fPIE -pie` (GCC) et ASLR le randomise ensuite. Par exemple, un exécutable compilé sans PIE apparaîtra dans `file` comme « not stripped, no PIE » tandis qu'un binaire PIE indiquera « pie executable » [9] [10]. Comme le note la documentation, sans PIE même avec ASLR le code reste à une adresse fixe [10], d'où l'intérêt de PIE pour la sécurité.
- **RELRO (Relocation Read-Only)** – Lorsqu'un programme utilise la liaison dynamique, il crée notamment une Global Offset Table (GOT) pour les appels aux fonctions externes. **RELRO** rend cette table *en lecture seule* après l'initialisation, empêchant certaines attaques de corruption de GOT. Il existe deux niveaux : *Partial RELRO* (par défaut souvent sur un binaire simple) bloque en lecture seule la plupart des sections après chargement, mais laisse l'initialisation différée possible. *Full RELRO* (activé avec `-Wl,-z,relro,-z,now`) interdit tout lien tardif et verrouille complètement la GOT. Checksec affichera « Partial RELRO » ou « Full RELRO » [11].
- **_FORTIFY_SOURCE** – C'est une option du compilateur (`-D_FORTIFY_SOURCE=2`) qui remplace certaines fonctions vulnérables par des versions de bibliothèque plus sûres (par exemple `memcpy`, `sprintf`, etc.) vérifiant la taille du buffer au runtime. Checksec mentionne « FORTIFY », mais l'activation effective dépend des fonctions utilisées. Cette protection n'était pas demandée explicitement dans votre exercice, mais on la note comme "activation par défaut dans beaucoup de binaires" lors de l'installation de certains paquets [12].

En résumé, sur un binaire Linux on voudra idéalement : **NX activé**, **canary présent**, **PIE activé** et **Full RELRO** pour maximiser la sécurité. On peut vérifier facilement avec `checksec --file=./test`, ou manuellement : par exemple `readelf -a test` (ou `readelf -W -h`) indique `Type: EXEC` ou `Type: DYN` (EXEC sans PIE, DYN avec PIE) [9] [10], et `readelf -l` affichera la section `GNU_STACK` (NX) et `GNU_RELRO`. La commande `file test` elle-même signale si le binaire est statique/dynamique, PIE/LSB, et « not stripped » [1] [13]. Par exemple,

```
$ checksec --file=./test
RELRO           STACK CANARY      NX            PIE
Partial RELRO   No canary found   NX enabled    No PIE
```

indique un binaire sans canari, sans PIE, avec NX et RELRO partiel.

Débordement de tampon (buffer overflow)

Un **dépassement de tampon** se produit lorsqu'un programme écrit *plus de données* qu'il ne peut en stocker dans un buffer, généralement situé sur la pile. En informatique, c'est un bug où « un processus écrit à l'extérieur de l'espace alloué au tampon, écrasant ainsi des informations nécessaires au processus » [14]. En pratique, cela veut dire qu'on peut écraser les variables locales situées juste après le buffer en mémoire. Sur la pile d'appel, les variables locales sont suivies par la sauvegarde du pointeur de pile (EBP/RBP) et enfin l'**adresse de retour** de la fonction. Écraser cette adresse de retour détourne le flux d'exécution : quand la fonction fait `ret`, le processeur saute vers l'adresse écrasée au lieu de continuer normalement. Comme l'explique la documentation, « l'attaquant peut ainsi contrôler le pointeur d'instruction après le retour de la fonction, et lui faire exécuter des instructions arbitraires » [15].

Par exemple, dans votre code `test.c` la fonction `vuln()` déclare un buffer de 0x20 octets et lit ensuite 0x100 octets de l'entrée standard :

```
char buffer[0x20];
fgets(buffer, 0x100, stdin);
```

Ici, il est évident que l'appel `fgets` lit jusqu'à 256 octets dans un espace qui n'en contient que 32. Les octets au-delà écraseront la pile au-delà du buffer. Comme le montre l'exemple de la page Wikipédia, si l'on copie 32 octets ou plus dans un buffer de 32 octets (puisque `fgets` n'arrête pas après 32), « la fonction strcpy continuera à copier le contenu de la chaîne au-delà de la zone allouée… c'est ainsi que les informations stockées dans la pile (incluant l'adresse de retour) pourront être écrasées » [16]. Dans le cas de votre programme, dès qu'on envoie 0x28 (40) octets à `fgets`, on atteint l'adresse de retour (32 octets de buffer + 8 octets pour RBP sauvegardé), ce qui provoque un plantage (“segfault”) et permettrait d'écraser la valeur de retour. En envoyant un peu plus d'octets on peut même écrire à l'emplacement de retour et forcer le `ret` de `vuln()` à sauter vers une adresse arbitraire (par exemple l'adresse de `gg()`).

Dans l'exemple pris de Wikipédia, on voit qu'en mettant 32 octets de “A” plus l'adresse de la pile, l'**attaquant peut placer du code malveillant sur la pile** et faire en sorte que le processeur l'exécute après le retour [17]. Concrètement, il suffit de déterminer le décalage exact à l'adresse de retour (ici 0x28) et d'y écrire l'adresse d'une fonction de son choix. Votre exercice a montré qu'après 40 (0x28) octets de « A » envoyés, le programme crashait, ce qui confirme que l'on a écrasé l'adresse de retour.

Exploitation et outils d'analyse (cheat sheet GDB)

Pour analyser et exploiter un débordement de tampon, on utilise des outils comme GDB et des commandes système. Voici quelques astuces clés :

- **Identification de l'architecture et du binaire** : utilisez `file ./test` ou `readelf -h ./test` pour vérifier 32/64 bits et dynamique/statique [1] [3]. Par exemple, la présence de « ELF 64-bit LSB » confirme un binaire 64 bits.
- **Vérifier les protections** : `checksec --file=./test` donne un résumé (RELRO, Canary, NX, PIE) [6]. Sinon, `readelf -l ./test | grep GNU_STACK` vérifie NX (s'il n'y a pas le flag `E` sur la pile) [8], `readelf -d ./test` (dit au lien) ou `file` indiquent la présence ou non de PIE et RELRO.
- **Lister les symboles et fonctions** : `readelf -s ./test` ou dans GDB la commande `info functions` liste les fonctions et leurs adresses. On peut aussi utiliser `objdump -d ./test`. Dans GDB, `p gg` affichera l'adresse de la fonction `gg` (par exemple `0x401176` dans votre cas) [18].
- **Emplacement du buffer dans la pile** : en lançant le programme dans GDB (`gdb ./test` puis `break vuln`, `run`), on peut afficher l'adresse du buffer avec `p &buffer` ou `info frame` pour voir le registre RSP/RBP au début de `vuln`. L'écart entre l'adresse du buffer et du retour donne le décalage. On peut aussi utiliser les commandes automatiques pour les motifs de débordement (par exemple `pattern_create` de Metasploit ou `pattern-offset` dans Pwntools) pour trouver l'offset exact du retour.
- **Contrôler l'exécution** : dans GDB on utilisera les commandes classiques – `break <fonction>` (ex. `b main` ou `b vuln`), `run` (ou `r`) pour lancer, `continue` (ou `c`), `step` / `next` pour avancer ligne par ligne. Lorsque le programme est arrêté, `x/40x $rsp` permet d'examiner la mémoire en format hex (ici 40 mots hexadécimaux à partir du pointeur RSP) [19].
- **Inspecter la pile et registres** : la commande `info registers` affiche les registres CPU (utile pour voir le RIP ou RSP) [20]. `info frame` donne des infos sur la trame courante (adresses). On utilise `x` (ex: `x/20xw <addr>`) pour voir la mémoire en hex ou ascii, et `print` (`p`) pour afficher des variables ou registres (par ex. `p $rip` ou `p 0xdeadbeef` pour tester). Ces commandes facilitent la compréhension de où on en est dans la mémoire.
- **Injection du payload** : pour tester le débordement, on peut envoyer directement des chaînes depuis le shell, par exemple `printf 'A%.0s' {1..40} | ./test` pour envoyer 40 « A ». L'outil `echo -e` ou un script Python `print('A'*40)` peuvent aussi être utilisés. Dans GDB on peut aussi exécuter le programme avec un argument ou rediriger un fichier.

En résumé, une **cheat sheet GDB** minimaliste pour ce type de challenge pourrait être :

- `gdb ./test` – lancer GDB sur le binaire.
- `b main` / `b vuln` – poser un breakpoint à l'entrée de la fonction vulnérable (ou au début).
- `r` – exécuter jusqu'au breakpoint.
- `info functions` – lister les fonctions connues (pour trouver `gg`).
- `p gg` – obtenir l'adresse de la fonction `gg`.
- `disas vuln` – désassembler `vuln` pour voir où est le buffer.
- `p &buffer` – afficher l'adresse du buffer local.
- `info frame` – voir la trame, notamment le registre base (RBP).
- `info registers` – voir les valeurs de registres (RSP, RIP, RBP, etc.) [20].

- `x/32xb $rsp` – inspecter 32 octets à l'adresse du pointeur de pile.
- `x/1gx <adresse>` – afficher un pointeur 64 bits (utile pour voir l'adresse de retour).
- `continue` ou `c` – reprendre l'exécution après breakpoint.

Protections à vérifier (via GDB ou checksec) : on a déjà vu NX, Canary, PIE, RELRO. GDB ne vérifie pas ces flags, mais `checksec --file` ou des commandes ELF le font. Par exemple, `readelf -W -s ./test | grep __stack_chk_fail` indique si le canari est présent (appel à `__stack_chk_fail`) [21].

En combinant ces outils, on peut répondre aux questions du QCM : le format ELF (64-bit, dynamiquement lié), la liaison du binaire (dynamique), l'état des symboles (non dépouillé si « not stripped »), et l'état des protections (NX, PIE, Canary, etc.). Par exemple, votre sortie de `checksec` affichait **Canary : ✘**, **NX : ✓**, **PIE : ✘**, **RELRO : Partial** – ce qui signifie « pas de stack canary, NX activé, pas de PIE, RELRO partiel » [6] [8].

En synthèse : un débordement de tampon (« buffer overflow ») permet d'écrire au-delà de la mémoire allouée, typiquement pour écraser l'adresse de retour sur la pile et détourner l'exécution. Les protections comme le canari, NX, PIE, RELRO sont là pour empêcher ou détecter ce genre d'attaque [14] [7]. Une bonne pratique consiste à compiler en activant toutes ces protections (`-fstack-protector-all`, `-z relro -z now`, `-fPIE -pie`) et à ne pas utiliser de fonctions non sécurisées (préférer `strncpy`, `fgets`, `snprintf`, etc. qui prennent la taille des tampons en compte [22]). Ces défenses ne sont pas infaillibles (ex. ROP pour contourner NX [23]), mais elles rendent les exploits bien plus difficiles. Enfin, maîtriser GDB (points d'arrêt, inspection mémoire, affichage des registres) est essentiel pour comprendre un binaire vulnérable et construire un exploit – cette cheat-sheet et les exemples ci-dessus devraient aider à réviser ces notions.

Sources : Documentation sur l'ELF et outils Linux [1] [3], articles sur les protections de sécurité des binaires [7] [8] [11], Wikipédia sur le buffer overflow [14] [15] [16], et tutoriels GDB [19].

---

[1] [6] [7] [8] [9] [11] [12] [21] Identify security properties on Linux using checksec | Opensource.com
https://opensource.com/article/21/6/linux-checksec

[2] [4] Static and Dynamic Linking in Operating Systems - GeeksforGeeks
https://www.geeksforgeeks.org/operating-systems/static-and-dynamic-linking-in-operating-systems/

[3] [5] [13] How static linking works on Linux | Opensource.com
https://opensource.com/article/22/6/static-linking-linux

[10] [18] [23] Stack buffer overflow - Wikipedia
https://en.wikipedia.org/wiki/Stack_buffer_overflow

[14] [15] [16] [17] [22] Dépassement de tampon — Wikipédia
https://fr.wikipedia.org/wiki/D%C3%A9passement_de_tampon

[19] [20] courses.cs.washington.edu
https://courses.cs.washington.edu/courses/cse484/20au/sections/slides/section_2.pdf