

Techniques et Outils d'Ingénierie Logicielle

...

John Gliksberg
(Bull SAS / Eviden / Groupe Atos)

Temporalité

- Cours 1:
 - Maîtriser un shell et ses automatisations
 - Avoir un environnement de développement productif
 - Savoir compiler un programme (lib, makefile, gcc, search paths)
 - Comprendre les phases de la compilation
- Cours 2:
 - Cycle de vie d'un programme
 - Documentation
 - Gestion de code source (Versionning) == GIT
 - Principe de forge (Issue, MR, fork, Pull)
 - Notion d'open Source
 - Gestion des conflits de code-source
- Cours 3:
 - **Débogage de programmes**
 - **Principe et structure des débogueurs**
 - **Utilisation de GDB**
 - **Structure d'un binaire**
 - **Structure d'un binaire lors de l'exécution**
 - **Débogage mémoire**
 - **Gestion d'erreur**

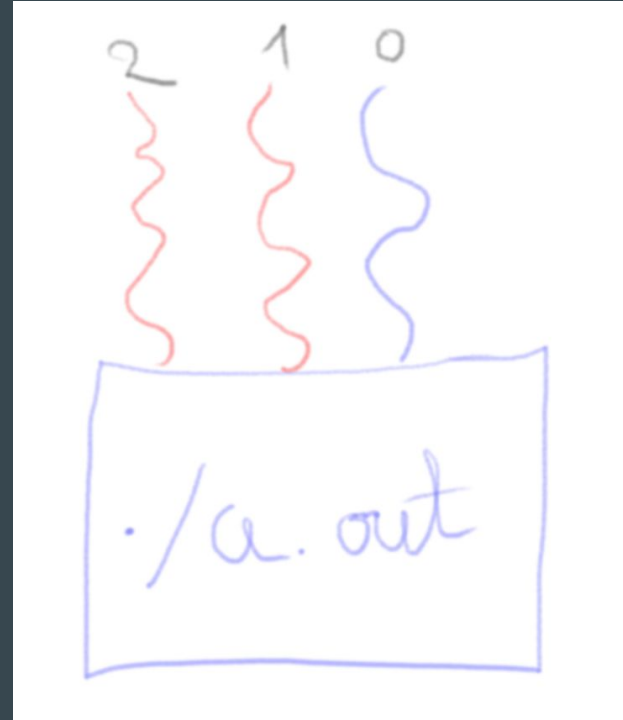
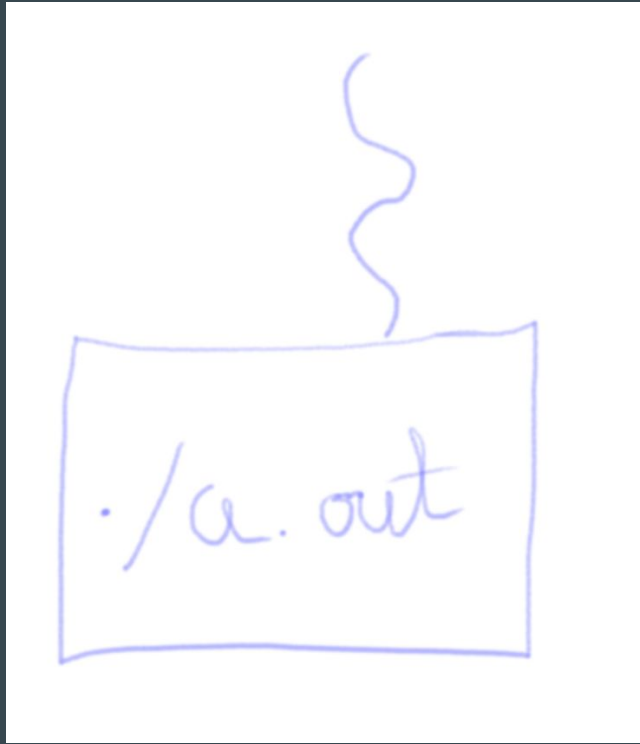
Pour récupérer les slides et exemples

git clone <https://github.com/trosh/TOI24>



Layout mémoire d'un programme

Processus canonique



La pile (*stack*)



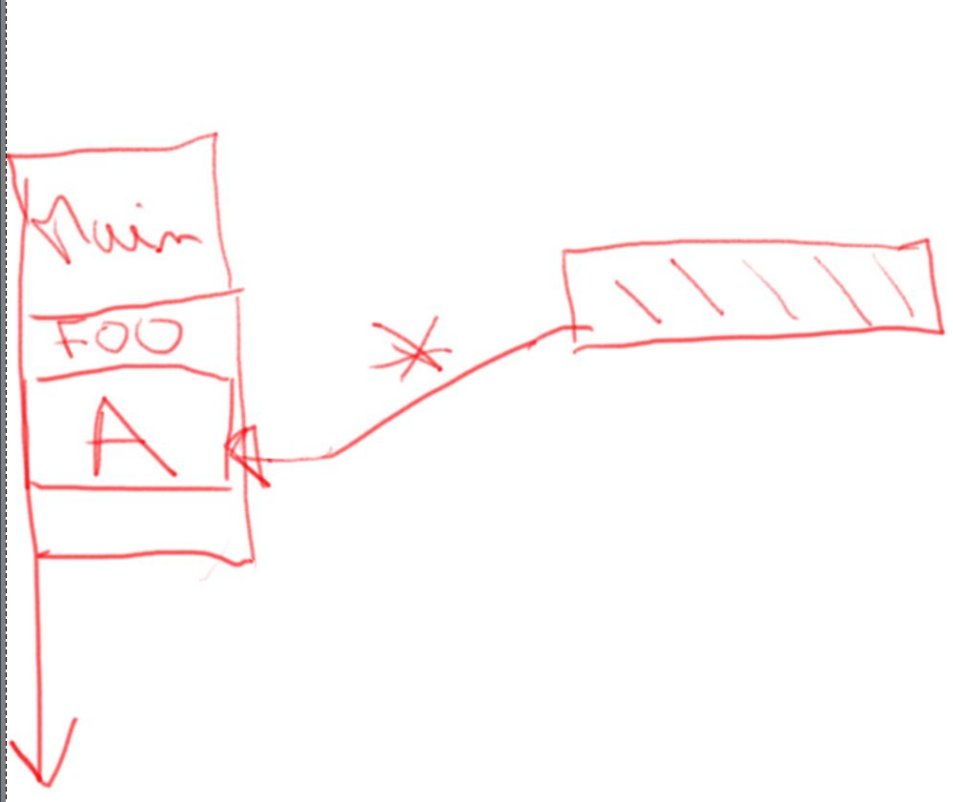
- La pile descend vers le bas
- Elle est de taille finie
- Chaque thread a sa pile
- On y empile les *frames* d'appel de fonctions
 - Registres et adresse de retour
 - Variables locales

Corruption de pile



- Pour chaque fonction, les variables sont côte à côte ; il est donc possible d'accidentellement les faire se modifier (dépassement de tampon)
- Si la pile devient trop grande : *stack overflow*
- Il est possible de corrompre l'adresse de retour de frame pour appeler un autre code
- Les corruptions de pile sont un problème de sécurité et difficiles à déboguer

Le tas (*heap*)



- Zone mémoire large
- Gérée par un allocateur :
 - `malloc(3)`
 - `free(3)`
- Une variable de la pile peut contenir une adresse du tas
- Espace paginé (4kB)

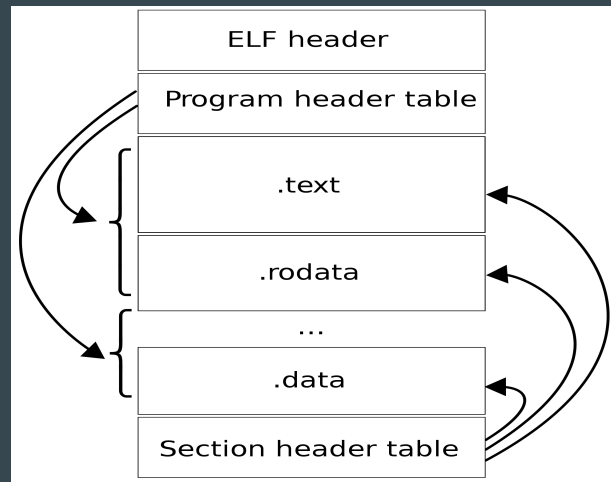
Corruption mémoire et le tas

- Dans le tas il est souvent plus difficile de voir les corruptions :
 - Pages par blocs de 4k
 - (pas de segfault pour overflow +1-1 par exemple).
 - Grandes zones et gestion de l'allocateur en interne sur les tampons
 - Il est possible de modifier des zones d'une autre variable accidentellement
 - Enfin, les fuites mémoires existent aussi

Le format ELF

Format ELF

- ELF = Executable and Linkable Format
- Décrit comment un binaire doit être représenté pour être compris par le lanceur de processus (ld-linux.so)
- Un programme contient beaucoup d'informations. Pour rester cohérent, il est segmenté en plusieurs sections
- Séquence magique : « **7F 45 4C 46** » = 7F « ELF »
- L'entête du ELF contient toutes les informations nécessaires à l'architecture (32/64 bits, endianness, ABI, type de fichier, jeu d'instruction...)



```
En-tête ELF:
Magique: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Classe: ELF64
Données: complément à 2, système à octets
Version: 1 (current)
OS/ABI: UNIX - System V
Version ABI: 0
Type: EXEC (fichier exécutable)
Machine: Advanced Micro Devices X86-64
Version: 0x1
Adresse du point d'entrée: 0x4003b0
Début des en-têtes de programme : 64 (octets dans le fichier)
Début des en-têtes de section : 6360 (octets dans le fichier)
Fanions: 0x0
Taille de cet en-tête: 64 (octets)
Taille de l'en-tête du programme: 56 (octets)
Nombre d'en-tête du programme: 9
Taille des en-têtes de section: 64 (octets)
Nombre d'en-têtes de section: 27
Table d'index des chaînes d'en-tête de section: 26
```

Format ELF

- **Program header** : stocke les informations nécessaires à la création de l'image du processus. Structure le programme d'un point de vue mémoire
- **Section header** : regroupe les informations nécessaires au bon fonctionnement du programme. Structure le programme d'un point de vue fonctionnel
- Le reste du ELF est composé de blocs d'instructions, indexées dans l'une et/ou l'autre de ces tables

En-têtes de section :

[Nr]	Nom	Type	Adr	Décala.	Taille	ES	Fan	LN	Inf	Al
[0]		NULL	0000000000000000	000000	000000	00	0	0	0	0
[1]	.interp	PROGBITS	0000000000400238	000238	00001c	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	0000000000400254	000254	000020	00	A	0	0	4
[3]	.note.gnu.build-id	NOTE	0000000000400274	000274	000024	00	A	0	0	4
[4]	.gnu.hash	GNU_HASH	0000000000400298	000298	00001c	00	A	5	0	8
[5]	.dynsym	DYNSYM	00000000004002b8	0002b8	000048	18	A	6	1	8
[6]	.dynstr	STRTAB	0000000000400300	000300	000040	00	A	0	0	1
[7]	.gnu.version	VERSYM	0000000000400340	000340	000006	02	A	5	0	2
[8]	.gnu.version_r	VERNEED	0000000000400348	000348	000020	00	A	6	1	8
[9]	.rela.dyn	RELA	0000000000400368	000368	000030	18	A	5	0	8
[10]	.init	PROGBITS	0000000000400398	000398	000017	00	AX	0	0	4
[11]	.text	PROGBITS	00000000004003b0	0003b0	000181	00	AX	0	0	16
[12]	.fini	PROGBITS	0000000000400534	000534	000009	00	AX	0	0	4
[13]	.rodata	PROGBITS	0000000000400540	000540	000010	00	A	0	0	8
[14]	.eh_frame_hdr	PROGBITS	0000000000400550	000550	000044	00	A	0	0	4
[15]	.eh_frame	PROGBITS	0000000000400598	000598	000118	00	A	0	0	8
[16]	.init_array	INIT_ARRAY	0000000000600e40	000e40	000008	08	WA	0	0	8
[17]	.fini_array	FINI_ARRAY	0000000000600e48	000e48	000008	08	WA	0	0	8
[18]	.dynamic	DYNAMIC	0000000000600e50	000e50	0001a0	10	WA	6	0	8
[19]	.got	PROGBITS	0000000000600ff0	000ff0	000010	08	WA	0	0	8
[20]	.got.plt	PROGBITS	0000000000601000	001000	000018	08	WA	0	0	8
[21]	.data	PROGBITS	0000000000601018	001018	000004	00	WA	0	0	1
[22]	.bss	NOBITS	000000000060101c	00101c	000004	00	WA	0	0	1
[23]	.comment	PROGBITS	0000000000000000	00101c	000058	01	MS	0	0	1
[24]	.symtab	SYMTAB	0000000000000000	001078	0005a0	18	25	41	8	
[25]	.strtab	STRTAB	0000000000000000	001618	0001c0	00	0	0	0	1
[26]	.shstrtab	STRTAB	0000000000000000	0017d8	0000f9	00	0	0	0	1

Clé des fanions :

W (écriture), A (allocation), X (exécution), M (fusion), S (chaînes), I (info),
L (ordre des liens), O (traitement supplémentaire par l'OS requis), G (groupe),
T (TLS), C (comprimé), x (inconnu), o (spécifique à l'OS), E (exclu),
l (grand), p (processor specific)

En-têtes de programme :

Type	Décalage	Adr. vir.	Adr.phys.	T.Fich.	T.Mém.	Fan	Alignement
PHDR	0x000040	0x0000000000400040	0x0000000000400040	0x0001f8	0x0001f8	R	E 0x8
INTERP	0x000238	0x0000000000400238	0x0000000000400238	0x00001c	0x00001c	R	0x1
[Réquisition de l'interpréteur de programme: /lib64/ld-linux-x86-64.so.2]							
LOAD	0x000000	0x0000000000400000	0x0000000000400000	0x0000b0	0x0000b0	R	E 0x200000
LOAD	0x000e40	0x0000000000600e40	0x0000000000600e40	0x00001dc	0x00001e0	RW	G 0x200000
DYNAMIC	0x000e50	0x0000000000600e50	0x0000000000600e50	0x0001a0	0x0001a0	RW	0x8
NOTE	0x000254	0x0000000000400254	0x0000000000400254	0x000044	0x000044	R	0x4
GNU_EH_FRAME	0x000550	0x0000000000400550	0x0000000000400550	0x000044	0x000044	R	0x4
GNU_STACK	0x000000	0x0000000000000000	0x0000000000000000	0x000000	0x000000	RW	0x10
GNU_RELRO	0x000e40	0x0000000000600e40	0x0000000000600e40	0x0001c0	0x0001c0	R	0x1

Format ELF

- `.text` : **code exécutable**
- `.data` / `.bss` : **données globales**
- `.rodata` : **constantes**
- `.tdata/.tbss` : Section de données thread-specific (TLS)
- `.got` : Table globale permettant d'avoir un accès indirect aux symboles globaux
- `.got.plt` : GOT pour fonctions dynamiques
- `.rel[a].*` : Symbole repositionnable, à résoudre avant le début du programme
- `.init` : prologue
- `.fini` : épilogue
- `.dynamic` : données utiles au loader pour charger les bibliothèques dynamiques
- `.dynstr` : Chaîne de noms des symboles globaux
- `.dynsym` : Table des symboles globaux
- `.symtab` : table de symbole
- `.c/dtors` : Stockage des routines `pre-main()`

Layout mémoire : *relocation*

```
$ cat /proc/self/maps
```

7fa293df8000-7fa293df9000	r--p	00000000	00:18	20589119	/usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa293df9000-7fa293e19000	r-xp	00001000	00:18	20589119	/usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa293e19000-7fa293e21000	r--p	00021000	00:18	20589119	/usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa293e22000-7fa293e23000	r--p	00029000	00:18	20589119	/usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa293e23000-7fa293e24000	rw-p	0002a000	00:18	20589119	/usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa293e24000-7fa293e25000	rw-p	00000000	00:00	0	
7ffc71429000-7ffc7144a000	rw-p	00000000	00:00	0	[stack]
7ffc715f0000-7ffc715f4000	r--p	00000000	00:00	0	[vvar]
7ffc715f4000-7ffc715f6000	r-xp	00000000	00:00	0	[vdso]

Voir `proc(5)`

Types d'instrumentation

Instrumentation par le compilateur

- Compiler le code instrumenté :
 - Ajoute les sondes lors de la compilation
 - Plus rapide que d'autres méthodes
 - Permet une instrumentation sélective
- Nécessite une recompilation :
 - `-fsanitize=address`
 - `-fsanitize=threads`
- Outils tels que ASAN présents dans les compilateurs modernes

Instrumentation à l'exécution

- Intercepter les appels du code :
 - Appel ptrace : utilisé par GDB
 - Bibliothèques instrumentées :
 - efence : malloc debugger
- Pas de recompilation
- Supporte les threads
- Nécessite les symboles de debug (compiler avec -g)
- Peut nécessiter un code non optimisé (-O0)

Virtualisation de l'exécution

- Intercepte l'ensemble des opérations
 - Accès mémoire
 - Allocations
- Dans le cas de Valgrind/Memcheck :
 - Exécution très lente ($\sim \times 1000$)
 - Pas de support des threads (exécution séquentielle)
- Pas besoin de recompiler

Utilisation de Valgrind



Fonctions de Valgrind

- *Memcheck* (défaut)
 - Détection des fuites de mémoire
 - Identifie les blocs mémoire alloués mais jamais libérés
 - Détection des accès mémoire invalides
 - Signale les accès à des zones mémoire non allouées ou libérées
 - ...
- *Helgrind*
 - Détection des *race conditions*
 - Repère les situations où plusieurs threads accèdent à la même ressource simultanément sans synchronisation
- *Cachegrind/Callgrind/Massif*
 - Profiling
 - Collecte des données sur l'utilisation du cache, de la prédiction de branchement, du tas

Approche de Valgrind

- Instrumentation : Valgrind modifie le binaire du programme à la volée pour insérer son propre code d'analyse
- Exécution : le programme instrumenté est exécuté par Valgrind.
- Analyse Dynamique : Valgrind surveille l'exécution du programme, collectant des informations sur les accès mémoire, les allocations et les libérations de mémoire, etc.
- Génération de Rapports : Valgrind produit des rapports détaillés sur les erreurs de mémoire, les fuites de mémoire, etc.
- Points forts de cette méthode
 - Précision : Valgrind surveille chaque instruction du programme, offrant une détection précise des erreurs de mémoire.
 - Indépendance de la Plateforme : Valgrind fonctionne sur plusieurs architectures et systèmes d'exploitation grâce à son approche de « virtualisation » de l'exécution du programme.
- Limitations
 - Overhead : l'instrumentation du code peut entraîner une surcharge significative des performances, rendant parfois l'exécution du programme plus lente.
 - Certaines erreurs peuvent échapper à la détection : malgré sa puissance, Valgrind peut ne pas détecter certaines erreurs de mémoire, notamment les cas complexes ou les comportements conditionnels.

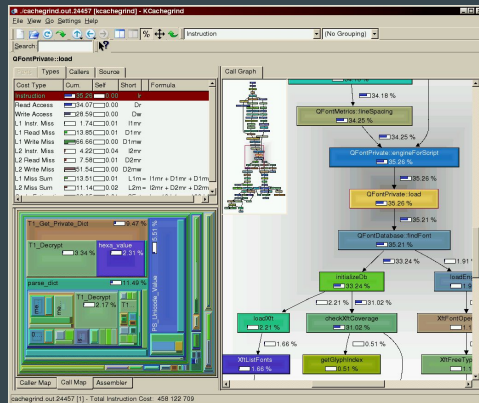
Utilisation Principale

- Sans recompiler mais avec overhead pour debug :
 - Débogage des accès mémoire et des allocations
 - Corruption de stack
 - Fuites mémoires
 - Pas de thread et gros overhead
- Alternative : ASAN (flag GCC/LLVM `-fsanitize=address`)
 - Doit recompiler
 - Moins d'overhead
 - Support des threads

Usage :

```
valgrind [--tool=...] a.out
```

Profilage simple avec KCachegrind



Utilisation principale

- KCachegrind est un outil de profilage graphique pour visualiser les données de profilage générées par des outils comme Callgrind
- Fonctionnalités de KCachegrind :
 - Visualisation graphique : affiche les données de profilage sous forme de graphiques interactifs
 - Analyse des performances : permet d'identifier les parties du code qui consomment le plus de temps CPU ou de mémoire
 - Navigation facile : permet de naviguer facilement à travers le code source pour localiser les goulots d'étranglement

Utilisation principale

Générer les données de profilage avec Valgrind :

```
$ valgrind --tool=callgrind ./my_program
```

Ouvrir les données dans KCachegrind :

```
$ kcachegrind
```

Lancez KCachegrind et ouvrez le fichier de données généré.

Analyser les performances et identifier les opportunités d'optimisation dans le code source.

Utilisation de ASAN

AddressSanitizer (ASAN)

- Qu'est-ce que AddressSanitizer (ASAN) ?
 - AddressSanitizer (ASAN) est un outil de détection des erreurs de mémoire développé par Google.
 - Conçu pour détecter les problèmes de mémoire courants tels que les débordements de tampon, les fuites de mémoire, etc.
- Fonctionnalités d'ASAN
 - Détection des dépassements de tampon : identifie les tentatives d'accès à des zones mémoire en dehors des limites allouées.
 - Détection des fuites de mémoire : signale les blocs mémoire alloués qui ne sont jamais libérés.
 - Détection des accès aux zones mémoire non initialisées : repère les accès à des variables non initialisées.

Utilisation

- Étapes pour utiliser ASAN
 - Intégration dans le processus de compilation :
 - Ajouter le drapeau de compilation `-fsanitize=address` lors de la compilation du programme.
 - Par exemple, avec GCC :
 - `gcc -fsanitize=address -o my_program my_program.c`
 - Exécution du programme :
 - Exécuter le programme comme d'habitude, sans aucune modification supplémentaire
 - Détection des erreurs de mémoire :
 - Pendant l'exécution, ASAN surveille l'accès à la mémoire et détecte les erreurs de mémoire telles que les débordements de tampon, les fuites de mémoire, etc.
 - Production de rapports d'erreurs :
 - ASAN produit des rapports détaillés sur les erreurs détectées, y compris les emplacements exacts dans le code source.

Utilisation

Compilation avec ASAN :

```
gcc -fsanitize=address -o my_program my_program.c
```

Exécution du programme :

```
./my_program
```

ASAN détecte les erreurs de mémoire pendant l'exécution et produit directement des rapports d'erreurs.

Débogage : le *debugger*

Erreurs et bogues communs

- Le débogage est crucial pour détecter et corriger une variété d'erreurs dans les programmes.

Les deadlocks

- Déboguer les deadlocks
 - Les deadlocks surviennent lorsque deux ou plusieurs processus ou threads restent bloqués indéfiniment, s'attendant mutuellement à libérer des ressources
 - Le débogage permet d'identifier les points de synchronisation mal conçus ou mal utilisés

L'erreur de segmentation

- Déboguer les *segmentation faults* / *segfaults* / SEGV
 - Les segfaults se produisent lorsqu'un programme tente d'accéder à une zone mémoire non autorisée
 - Le débogage permet de localiser précisément l'instruction responsable de l'accès mémoire incorrect

Les erreurs logiques

- Déboguer les erreurs logiques
 - Les erreurs logiques se produisent lorsque le comportement du programme ne correspond pas à ce qui est attendu, mais sans provoquer de crash ou d'erreur système
 - Le débogage est essentiel pour repérer et corriger ces erreurs subtiles qui peuvent altérer le bon fonctionnement du programme

La corruption mémoire

- Moins utile pour la corruption de mémoire
 - La corruption de mémoire peut entraîner des comportements imprévisibles et des crashes, mais elle est souvent difficile à détecter et à corriger via le débogage traditionnel
 - Des outils spécifiques comme Valgrind sont généralement plus adaptés pour détecter la corruption de mémoire
- En résumé:
 - Le débogage est un outil polyvalent pour identifier et résoudre une gamme variée d'erreurs dans les programmes
 - Comprendre les différents types d'erreurs et les techniques de débogage appropriées est essentiel pour développer des logiciels robustes et fiables

Appel système ptrace



Appel système ptrace

```
#include <sys/ptrace.h>
```

```
long ptrace(enum __ptrace_request request, pid_t pid,  
            void *addr, void *data);
```

Ptrace :

- Fonction système Unix pour le débogage
- Fonctionnalités :
 - Observation et le contrôle des processus
 - Lecture/écriture de mémoire
 - Manipulation des signaux
- On peut tracer un de ses sous-processus ou s'attacher à un processus existant

Les informations de debug (DWARF)

- Format DWARF :
 - Utilisé pour le débogage
 - Fournit des informations telles que les noms de variables, les types de données, etc.
 - Peut être stocké dans des fichiers séparés liés aux exécutables (paquets -dbg) et dans l'exécutable (-g)
- Utilisations :
 - Essentiel pour les outils de débogage, tels que gdb.
 - Facilite la localisation et la correction des erreurs dans le code.
- Avantages :
 - Réduit la taille des exécutables en séparant les informations de débogage.
 - Améliore l'efficacité du débogage en fournissant des informations détaillées sur le code.



Les informations de debug (DWARF)

- Utilisation de GDB :
 - Débogueur en ligne de commande pour les programmes C, C++, et autres langages
- Fonctionnalités principales :
 - Exécution pas à pas du code
 - Inspection des variables et de la mémoire
 - Suivi des appels de fonctions
- Commandes courantes :
 - *break* : Définit un point d'arrêt
 - *run* : Démarre l'exécution du programme
 - *print* : Affiche la valeur d'une variable
- Avantages :
 - Puissant pour le débogage en profondeur
 - Disponible sur la plupart des systèmes Unix-like

Lancement avec GDB

Lancement d'un programme avec GDB

- Ouvrir un terminal
- Naviguer vers le répertoire contenant le binaire du programme à déboguer
- Lancer GDB avec la commande :
 - `$ gdb ./a.out`
 - `$ gdb --args ./a.out args # Cas avec arguments inline`
- Configurer l'exécution
 - `(gdb) break main.c:27`
- Lancer le programme avec/sans arguments
 - `(gdb) run [<args> ...]`
- Interagir avec le programme durant la session
 - `(gdb) backtrace`
 - `(gdb) print s->obj_len`
 - `(gdb) watch s->objs[0].name`
 - `(gdb) continue`

GDB exécutera le programme jusqu'à ce qu'il rencontre un point d'arrêt ou qu'il se termine

Attach avec GDB

- Qu'est-ce que l'attachement dans GDB ?
 - L'attachement permet à GDB de se connecter à un processus, compilé avec -g, en cours d'exécution.
 - Utile pour déboguer des programmes déjà lancés ou des processus distants
- Trouver l'identifiant du processus que vous souhaitez attacher
 - En utilisant ps(1), pgrep(1), htop(1), ...
- Lancer GDB :
 - `$ gdb`
 - Utiliser la commande attach suivi de l'identifiant du processus :
 - `(gdb) attach <PID>`
- GDB se connectera au processus en cours d'exécution et vous pourrez commencer le débogage

Commandes de Base de GDB

- *r/run* Lancer le programme
- *b/break* Placer un point d'arrêt
- *c/continue* Reprendre l'exécution après un point d'arrêt
- *s/step* Exécuter la prochaine instruction, en entrant dans les sous fonctions
- *n/next* Exécuter la prochaine instruction, sans entrer dans les sous fonctions
- *p/print* Afficher la valeur d'une variable
- *q/quit* Quitter GDB

Commandes d'info

- *bt/backtrace* Affiche un backtrace
- *l/list* Affiche le code autour du point courant
- *disass* Désassemble le code autour du PC
- *i/info breakpoints* Afficher tous les points d'arrêt définis
- *i/info locals* Afficher les variables locales dans la fonction courante
- *i/info args* Afficher les arguments passés à la fonction courante
- *i/info registers* Afficher les valeurs des registres
- *display <variable>* Afficher continuellement la valeur d'une variable à chaque arrêt

Commandes de Navigation

- *s/step* Avancer en entrant
- *n/next* Avancer sans entrer
- *finish* Exécuter jusqu'à la fin de la fonction courante
- *u/until* Exécuter jusqu'à la fin de la boucle ou jusqu'à la ligne spécifiée
- *j/jump* <L> Sauter à une ligne spécifiée

Commande sur breakpoint

La commande « *command* » de GDB permet de programmer des actions à effectuer lorsqu'un point d'arrêt est rencontré

Syntaxe :

```
(gdb) break <...>  
Breakpoint X at 0x...: file ..., line ...  
(gdb) command <X>  
> commande1  
> commande2  
> ...  
> end
```

Commande sur breakpoint

Définir une commande à exécuter au point d'arrêt numéro 1 :

```
(gdb) command 1  
> print "Point d'arrêt atteint !"  
> backtrace  
> end
```

À chaque fois que le point d'arrêt numéro 1 est atteint, GDB affichera « Point d'arrêt atteint ! » suivi d'une trace de la pile (backtrace)

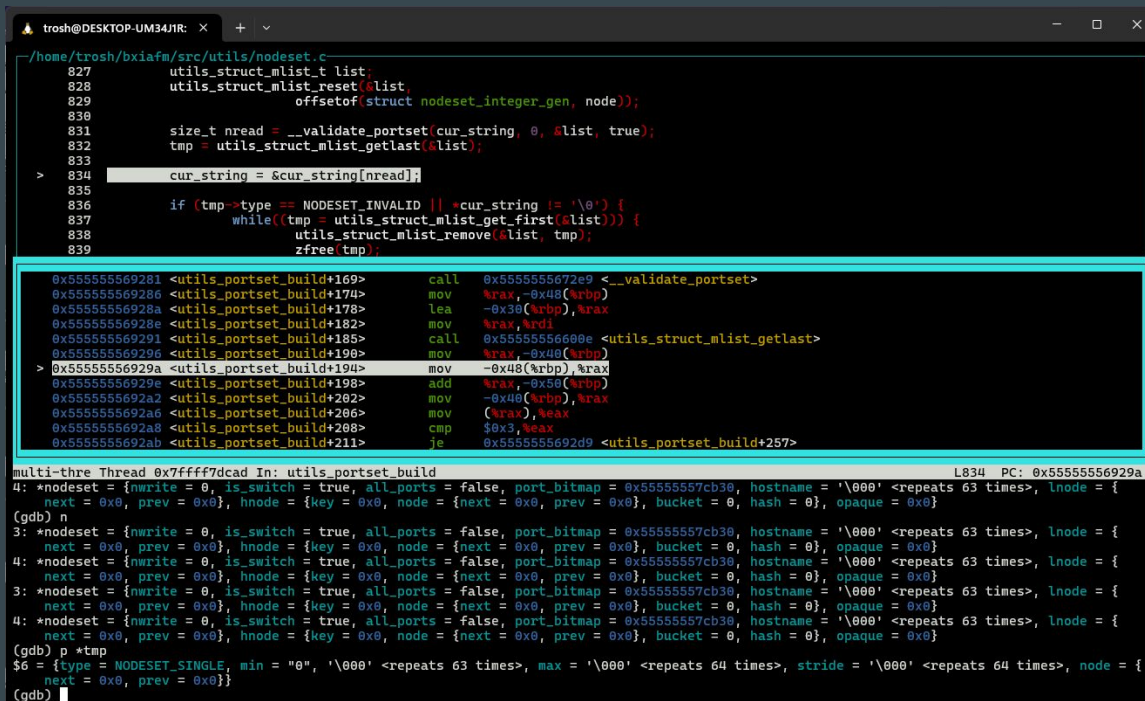
Mode TUI

GDB est fourni avec une interface « graphique »

(gdb) tui enable

(gdb) tui layout split

À comparer avec VSCode
et autres IDE



The screenshot displays the GDB TUI (Text User Interface) in a terminal window. The top pane shows the source code of `utils_struct_mlist_t` in `utils/nodeset.c`. The bottom pane is split into two sections: assembly code on the left and a multi-threaded stack trace on the right. The assembly code shows instructions like `call __validate_portset`, `mov $rax, -0x40(%rbp)`, and `lea -0x30(%rbp), %rax`. The stack trace shows multiple threads (e.g., `multi-thre Thread 0x7ffff7dcad In: utils_portset_build`) and their current state, including variables like `nwrite`, `is_switch`, `all_ports`, `port_bitmap`, `hostname`, `lnode`, `key`, `node`, `next`, `prev`, `bucket`, `hash`, and `opaque`.

```
trosh@DESKTOP-UM34J1R: X + -  
- /home/trosh/bxiafa/src/utils/nodeset.c  
827     utils_struct_mlist_t list;  
828     utils_struct_mlist_reset(&list,  
829                             offsetof(struct nodeset_integer_gen, node));  
830  
831     size_t nread = __validate_portset(cur_string, 0, &list, true);  
832     tmp = utils_struct_mlist_getlast(&list);  
833  
> 834     cur_string = &cur_string[nread];  
835  
836     if (tmp->type == NODESET_INVALID || *cur_string == '\\0') {  
837         while((tmp = utils_struct_mlist_get_first(&list))) {  
838             utils_struct_mlist_remove(&list, tmp);  
839             zfree(tmp);  
840  
0x555555569281 <utils_portset_build+169>    call    0x5555555672e9 <__validate_portset>  
0x555555569286 <utils_portset_build+174>    mov     %rax, -0x40(%rbp)  
0x55555556928a <utils_portset_build+178>    lea     -0x30(%rbp), %rax  
0x55555556928e <utils_portset_build+182>    mov     %rax, %rdi  
0x555555569291 <utils_portset_build+185>    call   0x55555556600e <utils_struct_mlist_getlast>  
0x555555569296 <utils_portset_build+190>    mov     %rax, -0x40(%rbp)  
> 0x55555556929a <utils_portset_build+194>    mov     -0x40(%rbp), %rax  
0x55555556929e <utils_portset_build+198>    add     %rax, -0x50(%rbp)  
0x5555555692a2 <utils_portset_build+202>    mov     -0x40(%rbp), %rax  
0x5555555692a6 <utils_portset_build+206>    mov     (%rax), %eax  
0x5555555692a8 <utils_portset_build+208>    cmp     $0x3, %eax  
0x5555555692ab <utils_portset_build+211>    je      0x5555555692d9 <utils_portset_build+257>  
  
multi-thre Thread 0x7ffff7dcad In: utils_portset_build L834 PC: 0x55555556929a  
4: *nodeset = {nwrite = 0, is_switch = true, all_ports = false, port_bitmap = 0x55555557cb30, hostname = '\\000' <repeats 63 times>, lnode = {  
    next = 0x0, prev = 0x0}, hnode = {key = 0x0, node = {next = 0x0, prev = 0x0}, bucket = 0, hash = 0}, opaque = 0x0}  
(gdb) n  
3: *nodeset = {nwrite = 0, is_switch = true, all_ports = false, port_bitmap = 0x55555557cb30, hostname = '\\000' <repeats 63 times>, lnode = {  
    next = 0x0, prev = 0x0}, hnode = {key = 0x0, node = {next = 0x0, prev = 0x0}, bucket = 0, hash = 0}, opaque = 0x0}  
4: *nodeset = {nwrite = 0, is_switch = true, all_ports = false, port_bitmap = 0x55555557cb30, hostname = '\\000' <repeats 63 times>, lnode = {  
    next = 0x0, prev = 0x0}, hnode = {key = 0x0, node = {next = 0x0, prev = 0x0}, bucket = 0, hash = 0}, opaque = 0x0}  
3: *nodeset = {nwrite = 0, is_switch = true, all_ports = false, port_bitmap = 0x55555557cb30, hostname = '\\000' <repeats 63 times>, lnode = {  
    next = 0x0, prev = 0x0}, hnode = {key = 0x0, node = {next = 0x0, prev = 0x0}, bucket = 0, hash = 0}, opaque = 0x0}  
4: *nodeset = {nwrite = 0, is_switch = true, all_ports = false, port_bitmap = 0x55555557cb30, hostname = '\\000' <repeats 63 times>, lnode = {  
    next = 0x0, prev = 0x0}, hnode = {key = 0x0, node = {next = 0x0, prev = 0x0}, bucket = 0, hash = 0}, opaque = 0x0}  
(gdb) p *tmp  
$6 = {type = NODESET_SINGLE, min = "0", '\\000' <repeats 63 times>, max = '\\000' <repeats 64 times>, stride = '\\000' <repeats 64 times>, node = {  
    next = 0x0, prev = 0x0}}  
(gdb)
```