Rapport de stage

DAMAS: Amélioration d'un outil d'instrumentation dynamique de binaires

DAUSSY Fabio
Sur les travaux de Camille LEBON
Encadrants : Pierre WILKE, Frédéric TRONEL
M1 Cybersécurité

Juillet 2024

Dans ce rapport, Nous allons faire un bilan de tout ce qui a été étudié, relevé et appris lors de ce stage de cinq semaines à CentraleSupélec dans l'équipe SUSHI. Le sujet était de reprendre le projet Damas d'une ancienne doctorante (Camille Lebon). Dans un premier temps, se l'approprier et le refaire fonctionner puis dans un second temps l'améliorer.

♦ Table des matières

♦ 1 Introduction

Damas est un projet écrit en Rust, se basant sur une API, nommée Sorry, qui permet de s'attacher à un processus. N'ayant aucune notion en Rust, mon stage commença par une remise à niveau dans le langage. Suivi d'une lecture approfondi de la thèse doctorale sur le projet [7] et du papier associé [8].

Damas est un prototype qui vise à sécuriser des processus. Il traite uniquement les binaires ELF x86_64 écrits en C et partiellement en C++. Pour étendre Damas au C++ sans bugs, Nous nous sommes intéressés au fonctionnement des exceptions en C++ [10, 2, 11, 12, 6, 1, 3, 4, 5, 9] à bas niveau et comment rendre leur gestions compatible dans Damas. Cette partie n'a, hélas, pas pu être amenée jusqu'à son bout.

Mais avant tout, nous allons introduire les concepts que Damas utilise et le fonctionnement de la sécurisation d'un processus.

♦ 2 Damas

■ 2.1 Définitions

Instrumentation dynamique de binaire (Dynamic Binary Instrumentation, DBI) :
C'est le procédé qui consiste à modifier les instructions d'un binaire pendant son exécution.

Bloc de base (Basic Block) :
C'est une séquence d'instructions sans branchement dans un programme.

```
mov $1, %rax
mov $2, %rbx
cmp %rax, %rbx
jge _foo
```

FIGURE 1 – Exemple d'un bloc de base

Graphe d'appels (Call Graph) :

Un graphe qui représente les appels de fonctions d'un programme. Chaque noeud représente une fonction. Un arc représente la possibilité de la fonction noeud à sa base d'appeler la fonction noeud à son bout.

Graphe de flot de contrôle (Control Flow Graph, CFG) :

C'est une représentation en graphe d'un programme. Chaque bloc de base du programme représente un noeud. Les arcs représentent la transition du flot de contrôle d'un bloc source vers un bloc destination.

Intégrité du flot de contrôle (Control Flow Integrity, CFI) :

C'est une propriété qui lorsqu'elle est vérifiée, assure le CFG du programme. Diverses techniques peuvent être utilisées pour s'approcher de la validité de la propriété. Damas, lui, contrôle les branchements indirects du programme.

Isolation des données de contrôle (Control-Data Isolation, CDI) :

Une technique qui assure la CFI du programme. Les branchements indirects sont remplacés par des branchements directs vers des toboggans.

Toboggan (Sled) :

Une séquence de tests d'égalités entre une adresse utilisée par un branchement indirect et une valeur immédiate ; ces valeurs sont les adresses possibles qu'un bloc de base peut appeler. Si une des conditions est respectée, un branchement direct sur l'adresse validée est pris. Si tous les tests sont effectués et qu'aucune égalité n'est relevée. Alors une erreur est générée.

```
if (fptr == addr1)
  call addr1;
else if (fptr == addr2)
  call addr2;
else if (fptr == addr3)
  call addr3;
```

FIGURE 2 – Exemple de toboggan dans une CDI

Dans l'exemple (2), *fptr* est une adresse qui est originalement appelée indirectement dans le programme. Dans une CDI, si *fptr* est modifié pendant l'exécution, alors il ne correspondra pas aux $addr_i$, et générera une erreur.

2.2 Contexte d'utilisation de Damas

Damas est donc un projet dont le but est de sécuriser les programmes/processus ELF x86_64. Ce qui peut distinguer Damas des autres projets est cette idée de sécuriser un processus sans avoir le code source, et surtout dynamiquement. Cela permet de sécuriser des programmes dont le code source n'est pas disponible. Mieux encore, de sécuriser des programmes dont le redémarrage pourrait être très coûteux, en temps, en performance voire en argent.

Ainsi, avant de sécuriser le processus il faut tout d'abord pouvoir s'y attacher.

2.3 API Sorry

Cette API, faisant aussi partie du projet, a pour objectif de permettre à Damas de s'attacher à un processus en tentant d'avoir le moins d'impact possible sur le temps d'exécution du processus. Sorry peut soit lancer un processus et s'y attacher, soit s'attacher à un processus existant grâce au PID. Sorry masque les appels à *ptrace* à l'aide d'une API simpliste qui fonctionne avec les structures suivantes :

- **TargetProcess** est une structure qui contient les informations sur le processus attaché. Une seule instance TargetProcess par processus analysé.
- **TargetController** est une structure qui, elle, contient les méthodes pour modifier la mémoire du processus. Il peut y avoir plusieurs TargetController pour un seul TargetProcess.
- **Buffer** est un trait Rust (en programmation orientée objet, cela serait semblable à une classe de base) qui permet d'écrire dans la mémoire du processus :
  - **RemoteBuffer** est une implantation du trait Buffer qui permet de modifier une partie de la mémoire déjà utilisée par le processus.
  - **CodeCache** est une implantation du trait Buffer qui permet de modifier une partie de la mémoire du processus qui n'est pas encore utilisée. Celle-ci est alors allouée.

```
let target = TargetProcess::from_pid(12345)?;
let ctrl = target.get_controller();
ctrl.load_map_file()?;

let option = ctrl.find_memory_map_entry(|entry| {
    entry.permissions.is_executable() &&
    entry.filename.unwrap_or("").contains("libc.so")});

match option {
    Some(entry) => println!("LibC loaded at: {:#x}",
        entry.start_addr),
    None => println!("No libC is loaded in the
        target process")
}
```

FIGURE 3 – Exemple d'utilisation de Sorry afin de savoir si le processus attaché utilise la libc.

Dans ce code (3), nous pouvons voir que le processus est attaché via son PID en créant une instance de TargetProcess (qui va contenir les informations sur le processus mais n'est pas encore modifiable. C'est en récupérant son TargetController que le processus a la possibilité de réaliser des modifications dessus (*target.get_controller*()). Dans l'exemple le contrôleur est utilisé pour récupérer des adresses des segments trouvables dans */proc/PID/maps*, il assure si la libc est chargée dans l'un d'eux et affiche son adresse si c'est le cas.

2.4 Le principe de table de délégations

Table de délégations :

C'est un toboggan (comme CDI) générée dynamiquement. Contrairement au CDI, la table de délégations ne contient pas les transitions exactes possibles du bloc appelant dû au manque d'informations sur le programme. Or, pour tenter de palier le manque d'informations, une table de délégations va vérifier une classe d'équivalence définie.

Dans Damas, il existe trois types de classes d'équivalence :

— Une classe d'équivalence pour les **appels indirects**. Ce sont tous les *call* présents au sein des blocs d'un programme dont l'opérande n'est pas une valeur immédiate ou relative à %rip.

— Une classe d'équivalence pour les **retours de fonctions**. Les instructions *ret* ne sont rien de plus que des sauts indirects sur l'adresse de retour de la fonction.

— Enfin, une classe d'équivalence par fonction. Pour garantir que les bornes d'un saut indirect au sein d'une fonction sont limités à cette fonction (i.e. il n'est pas autorisé de faire un saut indirect vers une autre fonction).

Le fait d'utiliser les tables de délégations ajoute une certaine flexibilité au contrôle du processus. En effet, certains comportements seront acceptés alors qu'ils ne respectent pas le CFG.

Prenons l'exemple des graphes d'appels (5, 4). La table de délégations des appels de fonctions indirects contrôlera que l'appel concerne bien une fonction. Mais, elle ne pourra pas contrôler si la fonction est normalement appelable par l'appelant comme le ferait la CDI théorique.

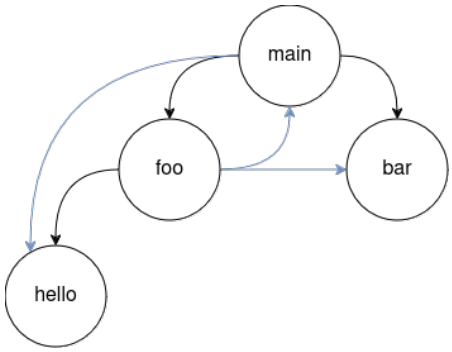

FIGURE 4 – Graphe d'appels respecté par la table de délégations des appels indirects. En bleu, les appels normalement non autorisés mais possible avec la table de délégations.

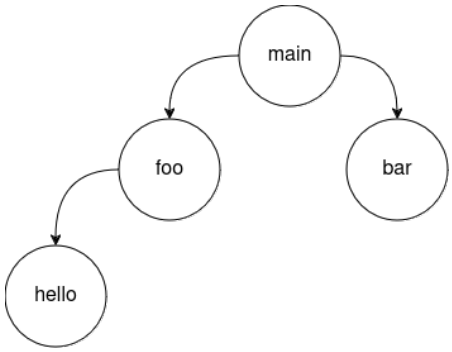

FIGURE 5 – Graphe d'appels d'une CDI respectant la structure du programme.

2.5 Vue d'ensemble de la structure de Damas

La première étape dans la sécurisation du processus consiste tout d'abord à récupérer les fonctions du programme. Les fonctions sont des ensembles de blocs de base. Pour ce faire, Damas utilise une bibliothèque pour exécuter du code python (PyO3) et ainsi déléguer cette tache à Angr [13] qui va lui renvoyer une structure contenant les fonctions et leurs blocs de base associés.

Ensuite, il faut parcourir tous les blocs et créer les instances des tables de délégations qui seront écrits en mémoire à l'aide d'un Buffer **Codecache**. Une fois les tables de délégations créées, il faut traduire tous les branchements indirects du programme en des branchements directs vers la table de délégations qui leur est associée. Or la traduction de ces sauts peut changer la taille des instructions qui les encodent au risque d'écraser les instructions qui suivent. Pour cela une nouvelle section *.text* est allouée en mémoire où de l'espace libre est ajouté entre chaque bloc de base pour changer les instructions des sauts indirects sans inquiétudes (6). Cette section est appelée *.secure_text*. De plus, en recréant une section, il faut aussi penser à adapter les branchements directs afin de ne plus ressauter dans la section *.text* !

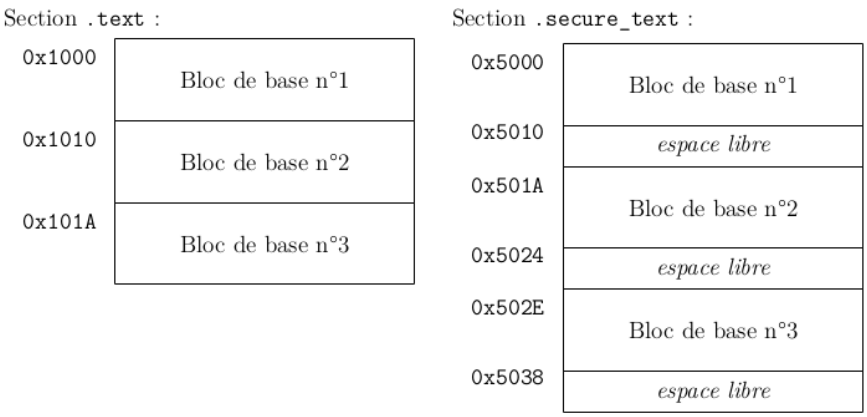

FIGURE 6 – Création de la nouvelle section .secure_text

La bijection entre les adresses du *.text* et le *.secure_text* se fait dans le code grâce à la structure **RelocationManager** qui contient les couples $(addr_{text}, addr_{stext})$. Les adresses opérandes dans les branchements directs sont rebasées selon la formule :

$$adr_{cible} = adr_{branchement} + taille_{branchement} + décalage \tag{1}$$

Où *décalage* est la variable qui prend en considération les espaces dans le *.secure_text*.

Une fois que la nouvelle section a été créée et le programme rebasé. Que les tables de délégations ont été aussi introduites dans la mémoire. Il ne reste alors plus qu'à préparer le programme pour qu'ils reprennent son exécution dans la section *.secure_text*. Pour cela, Damas regarde le pointeur d'instructions (%rip) courant et attend, si nécessaire, que celui-ci pointe une adresse du *.text*. Alors, ce pointeur est traduit et le processus analysé peut reprendre, ou commencer son exécution en sécurité.

■ 2.6 Construction des tables de délégations

Une fois la section *.text* rebasée en mémoire, il faut écrire en mémoire les instructions correspondant aux tables de délégations.

Dans le code, les tables de délégations sont écrites en instructions assembleur puis sont ensuite assemblées par un script (*as.sh*). Cela rend l'écriture des tables plus simple que d'écrire les instructions assembleur directement. Cela permet de gérer les sauts au sein de la table via des labels et laisser l'assembleur faire les calculs de décalage lui-même dans la table. Voyons alors comment construire les différentes tables.

▲ 2.6.1 Table de délégations principale (Main Dispatch table)

La table de délégations principale est la table qui s'occupe des appels de fonctions indirects. En assembleur cela correspondrait à tous les *call* dont l'opérande est un registre, une adresse calculée à partir d'un registre (7).

```
call %rdi
call %rax
call -8(%rbp)
; ...
```

FIGURE 7 – Appels indirects possibles

Or il faut savoir au préalable quel registre va contenir l'adresse à laquelle le programme veut sauter, c'est à dire l'adresse qu'il faut comparer dans la table de délégations principale. Damas introduit alors le concept de **registre de travail**. Ce registre va être celui qui contient l'adresse à comparer. Ainsi, pour mettre en place cela, avant de commencer les comparaisons dans la table. Le registre ou l'emplacement mémoire initialement utilisé va être transféré dans le registre de travail avant de sauter au début de la table. Dans la table de délégations principale, le registre de travail idéale avant de faire un *call* est **%rax** car il n'est pas utilisé à l'appel de la fonction (il est sollicité pour contenir la valeur de retour de la fonction appelée).

Supposons que %rdi contienne la valeur *0x1024* qui est l'adresse dans le *.text* de la fonction appelée, et que cette fonction ait pour adresse *0x2048* dans le *.secure_text*. Voici l'exemple de ce qui serait généré (Figure 8, 9, 10). Une fois dans la table, %rax est comparé avec tous les débuts de fonctions du programme et la table vérifie s'il correspond à l'un d'eux. Le cas échéant, le programme saute à l'adresse de la fonction rebasée et non l'originale.

```
_foo:
    ; ...
    call %rdi
    ; ...
```

FIGURE 8 – Appel sur %rdi non contrôlé.

```
_foo_stext:
    ; ...
    ; Le call est conservé pour
    ; maintenir le changement de frame
    call _main_dispatch_rdi_entry
    nop
    ; ...
```

FIGURE 9 – L'appel à %rdi est remplacé par un appel direct sur son entrée correspondante dans la table de délégations principale.

```
_main_dispatch_rdi_entry:
    mov %rdi, %rax
    jmp _main_dispatch_table
    ; ... peut-être d'autres entrées
_main_dispatch_table:
    ; ... les comparaisons avec le registre de travail.
```

FIGURE 10 – L'entrée de la table de délégations correspondante à l'appel sur %rdi, puis saut direct sur le début des comparaisons.

```
_main_dispatch_table:
    ; L'adresse comparée est celle du .text
    cmp %eax, $0x1024
    jne _next_case ; saut à la comparaison suivante
    ; l'égalité est vérifiée
    ; alors %rax va contenir sa nouvelle adresse
    mov $0x2048, %rax
    ; le saut est indirect sur %rax car il n'existe pas
    ; d'instructions jmp imm32
    jmp %rax
_next_case: ; cas suivant
    ; ...
```

FIGURE 11 – Exemple d'un cas de la table de délégations

▲ 2.6.2 Table de délégations des retours (Return Dispatch Table)

Contrairement à la table de délégations principale. La table de délégations des retours n'a pas besoin d'utiliser diverses entrées puisqu'au moment du retour d'une fonction, La valeur de retour se trouve sur (*%rsp*). Dans cette table, le registre de travail choisi est %r11. Au préalable, il faut conserver sa valeur actuelle avant de le transformer en registre de travail (13). %r11 est alors sauvegardé sur la pile à l'entrée dans la table des retours. Une fois une égalité trouvée, l'adresse du *.secure_text* sur laquelle sauter écrase celle du *.text*. N'oubliant pas aussi de dépiler 8 octets comme le ferait l'instruction initiale *ret* (12).

```
_ret_dispatch_table:
    ; Préambule, sauvegarde de %r11
    mov %r11, -0x8(%rsp)
    ; %r11 devient le registre de travail
    ; en récupérant l'adresse de retour
    ; initiale.
    mov (%rsp), %r11

    ; test de l'adresse initiale
    cmp %r11d, $0x1024
    jne _next_case

    ; saut vers l'adresse de retour rebasée
    mov $0x2048, %r11
    mov %r11, (%rsp)
    add $8, %rsp
    jmpq *-0x8(%rsp)
_next_case:
```

FIGURE 12 – Exemple du premier cas d'une table de délégations des retours

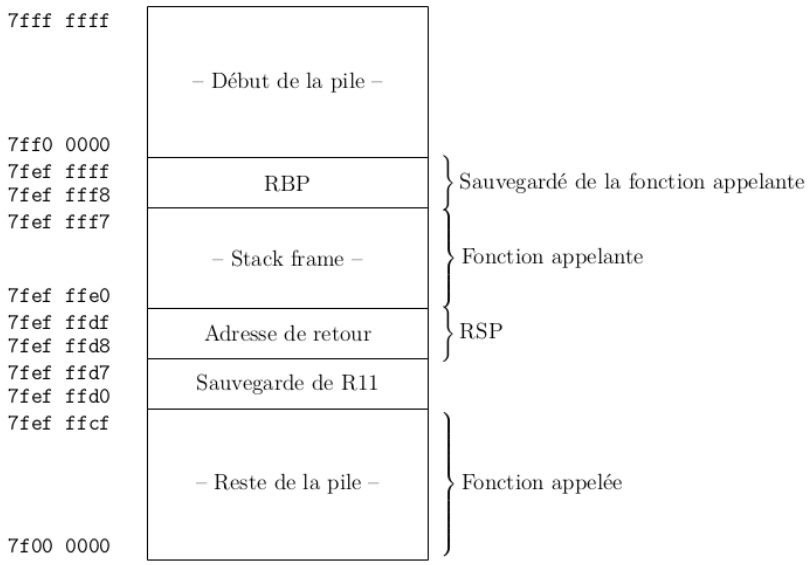

FIGURE 13 – Exemple de l'état de la pile à l'entrée dans la table de délégations des retours

▲ 2.6.3 Tables de délégations des sauts (Jump Dispatch Tables)

Les tables de délégations des sauts, comme dit dans l'introduction, vérifient les sauts indirects au sein d'une fonction. Elles assurent que la destination du saut dans une fonction concerne bien

la même fonction. Pour cela, au lieu de vérifier des adresses une à une comme fait dans la table principale et la table des retours, les tables de sauts vérifient si l'adresse du registre de travail (ici, %rax) est comprise dans les bornes des blocs de base de la fonction concernée (dans le *.text*). Si l'adresse est dans un bloc de la fonction alors le programme saute à l'adresse dans le *.secure_text* grâce à la formule :

$$cible_{rebasee} = cible + decalage \tag{2}$$

où le décalage correspond à celui entre les deux blocs :

$$decalage = bb_{stext} - bb_{text} \qquad \text{avec} \qquad bb_{stext} > bb_{text} \tag{3}$$

Supposons alors qu'une fonction contienne 2 blocs de base (bb_i correspond au bloc de base i).

```
TABLE_DELEGATIONS_SAUTS_FUNC_FOO:
    # sauvegarde de %rax
    # sauvegarde des flags
    SI bb_1.debut <= %rax < bb_1.fin ALORS
        rebase <- %rax + decalage1
        # restauration des flags
        # restauration de rax
        jump rebase
    SINON SI bb_2.debut <= %rax < bb_2.fin ALORS
        rebase <- %rax + decalage2
        # restauration des flags
        # restauration de rax
        jump rebase
    SINON
        # erreur
    FINSI
```

FIGURE 14 – Pseudo-code d'une table de délégations des sauts pour une fonction à deux blocs de base.

Dans l'exemple (14), Nous mentionnons le fait de sauvegarder les registres drapeaux sur la pile (avec *pushf*). Cela est nécessaire car ceux-ci pourraient être utilisés dans le corps de la fonction alors que les tests dans la table de délégations pourraient les altérer. Ainsi l'état de la pile lors de la sortie de la table est comme suit (15).

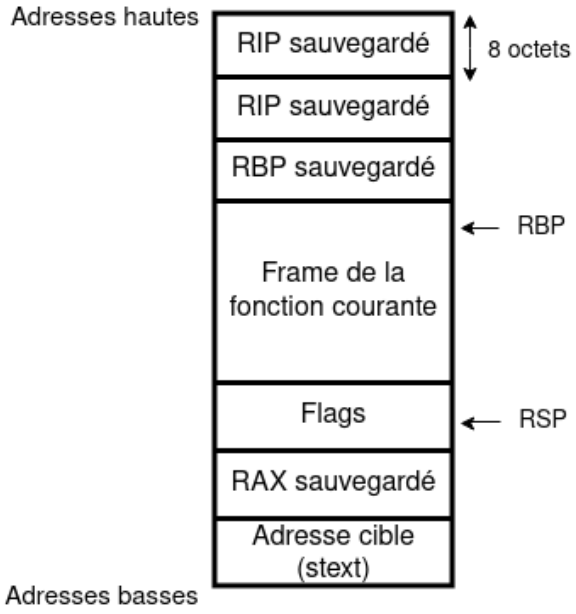

FIGURE 15 – État de la pile à la sortie de la table de délégations des sauts.

■ 2.7 Optimisation

Ainsi nous avons vu globalement le fonctionnement de la sécurisation d'un processus par Damas. Notons aussi que certaines fonctionnalités de Damas n'ont pas été détaillés mais sont importantes. Notamment, l'installation de compteurs dans les cas des tables afin de trier les cas des tables par ordres décroissants et permettre de réduire le temps d'exécution du programme. Ou encore, transformer les tables de délégations en arbres binaires de délégations afin de les parcourir en un temps logarithmique.

♦ 3 Contributions à Damas

Au départ je me suis investi sur la compréhension du code (et donc, de Rust). Puis avec les encadrants, nous avons commencé à dépoussiérer Damas. Le projet datant de 2022, il utilisait les *features* Rust, c'est à dire des fonctionnalités qui risquent d'être obsolètes. C'était le cas. Ainsi, nous avons mis à jour le projet en utilisant les *features* actuelles. Ensuite, il était primordiale de voir la cohérence entre ce qui est expliqué dans la thèse et ce qui est implanté.

■ 3.1 Ajout de débogage

Afin de vérifier cette cohérence, nous avons utilisé l'option existante *–loop* de Damas. Cette option permet à Damas de se détacher du processus et donc nous laisser la possibilité de nous y attacher avec un outil de débogage. Cette partie du code a été un peu améliorée. En effet, une fois détaché, Damas était dans une boucle infinie. Cela a été changé, dorénavant, Damas attendra un signal du processus analysé lors de son arrêt, et Damas sera alors à ce moment là stoppé. Il ne prendra plus de temps processeur pour rien.

En plus de ce changement, une nouvelle option a été ajoutée pour déboguer directement le processus depuis Damas. L'option *–gdb* permet alors à Damas, après la phase de rebasement, de reprendre directement l'exécution du programme contrôlé par *gdb*. C'est à dire qu'après avoir rebasé, Damas va utiliser l'appel système *execve* afin de lancer *gdb* avec comme argument le PID du processus analysé.

3.2 Collision dans les tables de délégations

Lors de l'analyse des tables de délégations et de leur génération. Une chose nous a dérangé : L'utilisation de registres 32 bits dans les tables alors que l'espace d'adressage du programme est sur 64 bits. L'affirmation précédente peut supposer que la création de table de cette manière peut générer des collisions au sein des comparaisons dans les tables. Essayons d'en générer. En reprenant l'exemple (11), nous remarquons que la comparaison du registre de travail avec une adresse originale possible se fait avec des valeurs 32 bits. Cela ne pose pas de problème pour des programmes qui sont petits. Mais si Damas venait à traiter des programmes conséquents, des collisions sont possibles.

Si l'on prend un exemple où les adresses à rebaser sont *0x1deadbeef* qui renvoie vers *0x2048* et *0x2deadbeef* qui renvoie vers *0xcafe* alors la table générée (16) contient une collision et seule le cas de *0x1deadbeef* sera pris. *0x2deadbeef* sera constamment ignoré.

```
    cmp $0xdeadbeef,%eax
    jnz _after_0xdeadbeef
    mov $0x2048,%rax
    jmpq *%rax
_after_0xdeadbeef:
    cmp $0xdeadbeef,%eax
    jnz _after_0xdeadbeef
    mov $0xcafe,%rax
    jmpq *%rax
_after_0xdeadbeef:
    ; error case
```

FIGURE 16 – Collision dans la table de délégations principale pour 0x1deadbeef → 0x2048 et 0x2deadbeef → 0xcafe

Pour éviter ce genre de collisions, il faut réaliser une comparaison de l'adresse *.text* sur 64 bits et %rax. Il n'existe pas d'instructions de comparaison entre un registre sur 64 bits et une valeur immédiate sur 64 bits. Or il existe l'instruction *cmp reg64, reg64* qui effectue une comparaison 64 bits entre deux registres 64 bits. Alors il faut songer à rajouter l'adresse *.text* dans un registre avant d'effectuer la comparaison de celui-ci et du registre de travail. Pour réaliser cela, nous avons mis en place un nouveau registre de travail qui est %r12. Tout comme %r11 dans la table de délégations des retours, ce deuxième registre de travail %r12 doit être sauvegardé sur la pile

en faisant attention à ne pas déranger le fonctionnement de base des tables. **Ce principe a été adapté à toutes les tables et aussi à la version en arbre binaire des tables de délégations.**.

```
    ; préambule: sauvegarde de %r12
    mov %r12, -0x8(%rsp)

    movabs $0x1deadbeef, %r12
    cmp %r12, %rax ; comparaison 64bits
    jnz _after_0x1deadbeef
    mov -0x8(%rsp), %r12
    movabs $0x2048, %rax
    jmpq *%rax
_after_0x1deadbeef:
    movabs $0x2deadbeef, %r12
    cmp %r12, %rax
    jnz _after_0x2deadbeef
    mov -0x8(%rsp), %r12
    movabs $0xcafe, %rax
    jmpq *%rax
_after_0x2deadbeef:
    ; error case
```

FIGURE 17 – Table de délégations principale corrigée pour 0x1deadbeef → 0x2048 et 0x2deadbeef → 0xcafe

Dans la correction (17) de la table en figure 16, le code généré marque bien la distinction entre les cas qui dépassent les 32 bits. Ainsi le code de la table comporte un peu plus d'instructions et prend plus de place en mémoire mais la justesse des branchements pris est assurée.

3.3 Main non protégé et heuristiques

Une fois les tables corrigées, nous avons réalisé quelques tests sur divers programmes. Nous avons alors remarqué qu'un programme dépouillé (dont les symboles ont été retirés) ne fonctionne pas avec Damas. En effet, Damas contrôlait les symboles du programme afin de chercher et de récupérer les bornes de la fonction *main*. En creusant, le but de cette recherche est de, lors de la traduction des sauts indirects, vérifier que le *ret* traduit n'est pas celui du *main*. En résumé, le saut indirect de retour *ret* du main est laissé tel quel et n'envoie pas vers la table de délégations des retours. Et donc, le main est vulnérable à une attaque de type stack buffer overflow.

De plus, si le programme est compilé avec la *libc* alors le programme passe par *__libc_start_call_main* pour appeler *main*. Les fonctions appelées depuis la PLT sont connues en mémoire au dernier moment. Si le programme analysé est lancé par Damas, nous ne savons pas lors du rebasement

quelle est l'adresse valide à laquelle le *main* doit retourner car *__libc_start_call_main* n'a pas encore été chargé.

C'est assez dommage si tout le programme cible est sécurisé sauf la fonction *main*. Alors, pour au moins assurer la sécurité du programme. Au lieu de ne rien faire lorsque Damas traite le *ret* du *main*. Le programme lance un appel système *exit*. Certes le programme n'est pas bien nettoyé comme le ferait la *libc* mais au moins, il est sécurisé.

A présent, il serait préférable que Damas puisse fonctionner sur des programmes dont les symboles ont été supprimés. Pour cela, nous avons mis en place des heuristiques pour que Damas puisse trouver le *main* (qui est la seule contrainte qui fait que Damas ne fonctionne pas sur les programmes dépouillés). Tout d'abord, Damas vérifie si le programme contient le symbole dynamique *__libc_start_call_main*.

— Si le symbole n'est pas présent, alors la première fonction appelée est le point d'entrée du programme. Dans le cas où le symbole n'est pas présent parce que la *libc* a été compilée statiquement dans le programme, cela ne pose pas de problème. Car nous connaissons dans ce cas là l'emplacement de *__libc_start_call_main* et Damas la traitera comme une fonction normale (qui passera par les tables). Dans les faits, peu importe la fonction, tant qu'elle est dans le *.text* du programme analysé, elle sera sécurisée par Damas. Donc un programme compilé statiquement ne pose pas de problème et nous pouvons considérer que la fonction principale (*main*) du programme est le point d'entrée.

— Si nous sommes dans le cas où le symbole est présent, cela signifie que le programme utilise la *libc* dynamiquement. Dans ce cas il est possible de retrouver l'adresse du *main* en analysant l'instruction juste avant l'appel à *__libc_start_call_main*. Elle correspond au passage de l'adresse du *main* en paramètre à la *__libc_start_call_main* soit dans %rdi. Cette instruction se trouve dans la fonction *_start* qui est le point d'entré d'un programme utilisant la *libc*.

  Une fois l'instruction repérée, il faut la traiter selon deux cas.

  1. Le programme est PIE. L'instruction correspond alors à un chargement dans %rdi de l'adresse du *main* relative au pointeur d'instructions %rip.

     ```
     0x1060:_start:
                 # ...
     0x1078:     lea     0xe4(%rip), %rdi # main
     0x107f:     call   *0x2f53(%rip)     # __libc_start_main@GLIBC_2.34
                 # ...
     ```

     Avec l'adresse, il est facile de retrouver les bornes de la fonctions car elles sont données par Angr dans les premières phases de la sécurisation.

  2. Le Programme est non-PIE. L'instruction correspond à un chargement dans %rdi de l'adresse absolue du *main*.

     ```
     0x4011020:_start:
                 # ...
     ```

```
0x401038:       mov     $0x401111,%rdi
0x40103f:       call    *0x2fab(%rip)   # __libc_start_main@GLIBC_2.34
                # ...
```

L'adresse, ici, est encore plus simple à récupérer car elle est directement.

En plus de ces heuristiques, nous avons tout de même laisser la possibilité à l'utilisateur de spécifier en argument l'adresse du *main* (*–main-addr*).

Damas fonctionne alors pour des programmes C qui sont dépouillés, mais quand est-il des programmes C++ ?

♦ 4 Le comportement de Damas sur des programmes C++

■ 4.1 Les Vtables C++

Le C++ est un langage de programmation orienté objet. Ainsi les Structures classiques sont étendues à des Classes. Dans ce paradigme, les méthodes sont des fonctions spécifiques à une classe donnée. De plus, la notion d'héritage de classe, permet à une classe fille de réécrire ou non les méthodes d'une classe mère. Certaines classes existent mais ne peuvent pas être instanciées. Elles servent uniquement de base à d'autres classes filles. Ce sont les classes abstraites. Il est alors possible en C++ de parcourir un tableau d'instances de classes filles et de les conserver dans un pointeur de la classe dont elles héritent (la classe mère). Ce comportement, le polymorphisme, est très utile pour les développeurs. Dans le code (19), la boucle *for* arrive bien à faire la distinction entre les instances dans le tableau, appelle dans le premier cas la méthode *description* de la classe Etudiant et le second cas la méthode *description* de la classe Prof. Le diagramme de classe du programme (18) indique en rouge un attribut caché qui permet d'assurer l'appel à la bonne méthode correspondante. Le vptr pointeur virtuelle, indique une table qui est commune à toutes instances d'une même classe : La Vtable. Cette table contient toutes les méthodes appelables par une classe. Ainsi, pour appeler une méthode, le compilateur passe par la vtable qui est pointée par l'objet concerné.

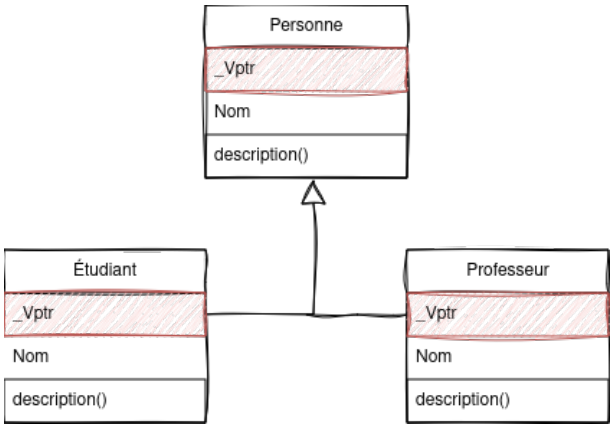

FIGURE 18 – Exemple d'un diagramme de classes montrant le vptr.

```
Personne * pers[2];
pers[0] = new Etudiant("Tim");
pers[1] = new Prof("Alain");

for(int i = 0; i < 2; i++){
    // Comment le compilateur sait quelle méthode appeler ?
    pers[i]->description();
}
```

FIGURE 19 – Exemple de code utilisant le polymorphisme

La façon dont Damas gère cela est assez simple, il considère les méthodes comme des fonctions et va donc les placer au niveau de la table de délégations principales. Dans le principe fondamentale de Damas, cela reste correcte puisque une méthode est bien une fonction. Or, nous pourrions être critique sur le fait qu'il n'y a aucune différence qui est faite entre les classes. En effet, si un attaquant parvient à modifier le vptr d'un objet, et le fait pointer sur une autre Vtable que la sienne, il n'y aura aucun problème au niveau de la table de délégations. Donc une instance de classe A pourrait appeler une instance de classe B sans se faire relever par Damas, ce qui va un peu à l'encontre du paradigme de la programmation orientée objet.

Malgré cela, la sécurité apportée par Damas sur l'appel indirect sur *vptr + décalage* reste suffisante, voire satisfaisante. De ce fait, ce problème n'a pas été creusé plus loin.

4.2 Les exceptions C++

Une autre partie à laquelle nous nous sommes intéressés est celle concernant la gestion des exceptions en C++. En lançant le programme (20) dans Damas, nous avons alors remarqué que l'exception "error" n'était pas attrapée par le *catch* ce qui faisait *abort* le programme. Il a fallu alors se plonger sur la compréhension des exceptions en C++ afin de tenter de trouver une solution à ce problème.

```
void stuff() {
  puts("Begin\n");
  throw "error";
  puts("End\n");
}

int main() {
  try {
    stuff();
  }
  catch(char const* e) {
    printf("%s\n", e);
  }
  return 0;
}
```

FIGURE 20 – Exemple d'un programme C++ attrapant une exception

▲ 4.2.1 Côté utilisateur

Lorsque nous souhaitons gérer une exception dans un programme C++. Nous pouvons assigner un bloc *try* qui va exécuter le code normale du programme. Il est possible que ce bloc ou les fonctions qui y sont appelées génère une erreur à l'aide du mot clé *throw*. Une erreur peut être de n'importe quel type. Ainsi, un ou plusieurs autres blocs de code *catch* sont associés au *try* qui les précède et s'exécute si l'erreur envoyée par *try* leur correspond.

Mais comment le bloc *catch* correspondant est trouvé alors que le *throw* peut provenir d'une autre fonction dont la *frame* est plus basse dans la pile.

▲ 4.2.2 Dérouler la pile (Stack Unwinding)

Pour répondre à l'interrogation précédente, il faut comprendre le principe de déroulement de pile, ou Stack Unwinding. Lorsqu'un *throw* est lancé, cela appelle en réalité une fonction du même nom qui va lancer le processus de déroulement de la pile via *unwind_raise_exception* qui se déroule en deux étapes [6].

1. La première phase (*_UA_SEARCH_PHASE*) consiste à remonter la pile en cherchant le bloc *catch* correspondant à l'erreur envoyée. Si toute la pile est remontée et qu'aucun bloc ne correspond, la fonction *std : :terminate* est appelée et le programme *abort*. C'est ce qu'il se passait avec Damas dans le code 20. Si le *handler* est trouvé, la deuxième phase est lancée.
2. La deuxième phase (*_UA_CLEANUP_PHASE*) consiste à parcourir à nouveau la pile jusqu'au *handler* mais en désallouant les *frames* qui ne sont plus nécessaires dans le programme. Puis une fois arrivée au *handler*, Exécuter le bloc correspondant.

Toutes ces manipulations sont possibles grâce à des informations de *debug* ajoutées par DWARF [1]. DWARF est un standard qui introduit un format de *debug* pour les programmes. Il assure, pour les programmes C++, l'état des registres pour toutes les instructions, il ajoute des instructions sur comment dérouler la pile, il indique les adresses des *catchs* associés aux *frames* et plus. Il existe en mémoire, dans la section *.eh_frame* un tableau contenant ces informations. C'est ce tableau qui est utilisé lors du déroulement de la pile.

▲ 4.2.3 .eh_frame_hdr et .eh_frame

Encoding	Field
unsigned byte	version
unsigned byte	eh_frame_ptr_enc
unsigned byte	fde_count_enc
unsigned byte	table_enc
encoded	eh_frame_ptr
encoded	fde_count
	binary search table

FIGURE 21 – Structure de .eh_frame_hdr

0x10 foo	@fde_foo
0x20 bar	@fde_bar
0x35 main	@fde_main

FIGURE 22 – Exemple d'une table de recherche binaire d'un ._eh_frame_hdr

Pour accéder à ce tableau lors du déroulement de la pile, le *loader* possède un pointeur sur le début de la section *.eh_frame_hdr* qui est l'en-tête de *.eh_frame*. Cet en-tête (21) contient plusieurs champs, notamment des les champs **_enc* qui spécifient comment les octets suivants sont encodés (ce qui est assez laborieux pour un humain de décortiquer à la main). La table de recherche binaire (22) est une séquence de couple triée ou le premier élément correspond à l'adresse d'une fonction dans le programme et le second correspond au décalage dans *.eh_frame* pour atteindre son FDE.

Le FDE (Frame Description Entry) est une entrée dans la table *.eh_frame* qui va contenir des informations sur la *frame* de la fonction qu'il désigne (les adresses, les zones mémoires où sont stockés les registres...). Il y a un FDE par fonction. Plus précisément, il y a deux types d'entrées dans *.eh_frame*. Les FDE et les CIE (Common Information Entry), les CIE concerne un ou plusieurs FDE. Certains FDE peuvent contenir des informations similaires (par exemple sur les sauvegardes des registres). Alors, afin d'éviter la redondance d'informations et de gagner de la place, ces informations ont été factorisées dans les CIE (23)

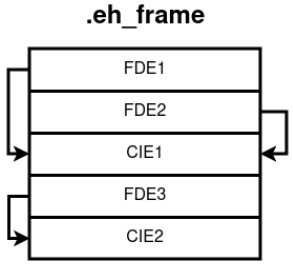

FIGURE 23 – Schéma de la section .eh_frame

Length	Required
Extended Length	Optional
CIE Pointer	Required
PC Begin	Required
PC Range	Required
Augmentation Data Length	Optional
Augmentation Data	Optional
Call Frame Instructions	Required
Padding	

FIGURE 24 – Structure d'un FDE

Maintenant que nous avons vu globalement le fonctionnement des exceptions, revenons à notre problème. Les exceptions ne sont pas prises en compte sur le code rebasé dans la *.secure_text*. D'après la structure des exceptions, les bornes des fonctions sont notées dans les FDE (24) et sont probablement vérifiées lors du déroulement de la pile. De même, pour l'en-tête qui utilise l'adresse du début de la fonction pour l'associer au FDE correspondant.

▲ 4.2.4 Réécrire .eh_frame_hdr et .eh_frame

L'étape première pour tenter de résoudre le problème des exceptions est de s'assurer que les bornes des fonctions retrouvées dans les FDE correspondent à celle de *.secure_text*. L'étape d'après est de faire en sorte que le code du *catch* exécuté soit aussi celui de la section *.secure_text* et pas celui de la section *.text*.

Alors, nous avons créer un module sur Damas qui permet de lire la section *.eh_frame_hdr* (grâce à un **RemoteBuffer**) et de la stocker dans une structure EhFrameHdr. Le module est conçu a une structure pour gérer plus tard tous les types d'encodage utilisés par l'en-tête. Mais pour le moment, les types d'encodages utilisés sont ceux de *gcc / g++*. Grâce aux premiers champs de l'en-tête, la structure fabrique un vecteur correspondant à la table de recherche binaire et contient alors les couples (*fonction, fde*). Une méthode pour traduire les adresses du *.text* en *.secure_text* est appliquée sur chaque adresse du vecteur sauf celles appartenant à la plt (donc *.plt, .plt.got, .plt.sec*). Après la traduction, le vecteur est trié. Enfin, la structure est réécrite dans la section via le **RemoteBuffer** en tenant compte des changements effectués.

L'en-tête étant traité, les champs *pc_begin et pc_range* des FDE doivent aussi être changés et adaptés à la section *.secure_text*. Où *pc_begin* correspond à l'adresse de début de la fonction du FDE et *pc_range* correspond à la taille de la fonction en octets (des instructions).

Malheureusement, même avec ces changements, le programme rebasé par Damas ne parvient pas à attraper les exceptions. Nous n'avons pas eu le temps de plus avancer car le stage arrivait à sa fin.

♦ 5 Perspectives

Pour la suite, il faut reconsidérer le problème des exceptions en C++. Comme Damas est un programme qui s'attache à des binaires dynamiquement. Réécrire par dessus les sections de gestion d'exceptions ne suffit pas. Supposons que Damas s'attache à un programme déjà en cours d'exécution. Alors dans la pile il y aura des informations sur les *frames* des fonctions du *.text*. Et par la suite il y aura aussi des *frames* de fonction du *.secure_text*. Les deux seront mélangées. Il est alors nécessaire que *.eh_frame* double sa taille pour y accueillir les deux types de fonction. Or, doubler la taille de *.eh_frame* en mémoire n'est pas si simple, car il n'est pas possible de prolonger l'actuelle. Elle est complètement bornée par deux autres sections. Même problème pour *.eh_frame_hdr*.

De ce fait, il faudrait allouer de la mémoire pour réécrire une section *.eh_frame_hdr* et *.eh_frame*. Ce qui signifie aussi qu'il faut s'intéresser à comment le pointeur sur *.eh_frame_hdr* est passé au

loader. Et s'il est possible de le modifier dynamiquement pour pointer vers la nouvelle.

♦ 6 Conclusion

Mon retour sur cette expérience est positif. Certes, c'est un exercice que j'ai trouvé particulièrement difficile. Se plonger pendant plusieurs semaines sur un même sujet peut être parfois décourageant lorsque des obstacles sont rencontrés. Mais les petites réussites sont plus gratifiantes que d'habitude dans ces moments là. Les sujets traités sont intéressants et techniques. J'y ai appris beaucoup de notions. En tout cas, j'ai tenté de répondre au mieux aux problématiques que rencontrait Damas, et ai résumé le plus synthétiquement possible ce que j'ai appris, lu, compris lors de ce stage.

♦ Références

[1] DWARF Debugging Information Format Committee. Dwarf debugging information format version 5, 2017. URL : https://dwarfstd.org/doc/DWARF5.pdf.

[2] Darius Engler. How c++ exceptions work, 2023. URL : https://www.lre.epita.fr/news_content/SS_summer_week_pres/Darius_Engler_cxx_exception.pdf.

[3] Linux Foundation. Dwarf extensions. URL : https://refspecs.linuxfoundation.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic/dwarfext.html.

[4] Linux Foundation. Exception frame. URL : https://refspecs.linuxfoundation.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html.

[5] Itanium. Exception handling tables. URL : http://itanium-cxx-abi.github.io/cxx-abi/exceptions.pdf.

[6] Itanium. Itanium c++ abi : Exception handling. URL : https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html.

[7] Camille Le Bon. *Analyse et optimisation dynamiques de programmes au format binaire pour la cybersécurité*. Theses, Université Rennes 1, July 2022. URL : https://theses.hal.science/tel-03906421.

[8] Camille Le Bon, Erven Rohou, Frédéric Tronel, and Guillaume Hiet. DAMAS : Control-Data Isolation at Runtime through Dynamic Binary Modification. In *SILM 2021 - Workshop on the Security of Software / Hardware Interfaces*, 2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), pages 86–95, digital event, Austria, September 2021. URL : https://hal.science/hal-03340008, doi:10.1109/EuroSPW54576.2021.00016.

[9] Kévin Lesénéchal. Unwinding the stack the hard way, 2023. URL : https://lesenechal.fr/en/linux/unwinding-the-stack-the-hard-way.

[10] Martin Balao. Understanding the .gcc_except_table section in elf binaries (gcc), 2019. URL : https://martin.uy/blog/understanding-the-gcc_except_table-section-in-elf-binaries-gcc/.

[11] Francesco Zappa Nardelli. Tales from binary formats, 2018. URL : https://entropy2018.sciencesconf.org/data/nardelli.pdf.

[12] James Oakley. Exploiting the Hard-Working DWARF : Trojan and exploit techniques with no native executable code. In *5th USENIX Workshop on Offensive Technologies (WOOT 11)*, San Francisco, CA, August 2011. USENIX Association. URL : https://www.usenix.org/conference/woot11/exploiting-hard-working-dwarf-trojan-and-exploit-techniques-no-native-executable.

[13] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK : (State of) The Art of War : Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.