05/2016

Debugger Windows

BOKHABRINE
Wissam

RYMER
Julie

Partie 1 : Fonctionnement d'un debugger

Un **debugger** est un programme qui exécute un autre programme cible aussi appelé **debuggee**.

Il est utilisé pour contrôler / analyser l'exécution d'un debuggee à l'aide de fonctions de contrôle qui peuvent stopper l'exécution du debuggee à chaque instruction et de fonctions d'affichages qui permettent de consulter la mémoire du debuggee (registres, pile, ...) et ceci à n'importe quelle étape de son exécution.
Il fonctionne sur le principe d'une boucle principale dans laquelle il va recevoir plusieurs événements provoqués par l'exécution du programme cible. À la réception d'un événement, le debugger effectue l'action correspondante à l'aide de ses fonctions de contrôle et d'affichage.

Les fonctions de contrôle d'un debugger peuvent permettre de faire :

-Un breakpoint, qui permet de stopper l'exécution d'un programme à une adresse précise.

-Un breakpoint conditionnel, qui permet de stopper l'exécution d'un programme lorsqu'il rempli une condition.

-Un step over, qui exécute la prochaine instruction du debuggee sans rentrer dans les sous-routines.

-Un step into, qui exécute la prochaine instruction du debuggee et rentre dans les sous-routines.

Les fonctions d'affichage d'un debugger peuvent permettre de montrer :

-Une zone mémoire du programme cible.

-Les valeurs des registres du microprocesseur.

-Le code assembleur ou le code source.

Un debugger est généralement utilisé pour isoler la source d'un bug dans un programme (d'où son nom).

Mais il peut aussi être utilisé pour d'autres raisons comme comprendre le fonctionnement d'un programme inconnu.

Partie 2 : Implémentation.

Nous avons fait un debugger Windows x86-32 en C++ qui tourne sans interface graphique sur l'Invite de Commandes. Il interagit avec les réglages de la console pour une meilleure esthétique. La bibliothèque « Capstone » a été utilisée pour afficher le binaire sous forme de code désassemblé.

La boucle principale

La boucle principale du debbugger est la fonction 'EnterDebugLoop' qui gère tout les événements que l'exécution du programme cible provoque.

Elle traite les événements sous deux grands switch :

Le premier est le switch qui englobe les événements suivants :

-Création du processus cible qui informe de la création du debuggee.
-Fermeture du processus cible qui informe que le debuggee a fini son exécution.
-Création d'un thread qui informe qu'un thread a été crée.
-Fermeture d'un thread qui informe qu'un thread a fini son exécution.
-Chargement/Déchargement d'une DLL qui informe qu'une bibliothèque de liaison dynamique a été chargée ou déchargée.

Le second switch est celui qui traite les exceptions suivantes :

-Control+C qui informe que le debuggee a reçu ce signal.
-Breakpoint qui apparaît lorsque le programme cible a lu un 0xCC. C'est ici que la commande step over et une partie des breakpoints sont traités.
-Single_step qui apparaît lorsque le trap flag a été mis à 1 et qui permet de traiter le reste de la commande breakpoint et la commande step into.

Après avoir traité l’événement reçu par le debuggee, la fonction continue l'exécution du debuggee qui avait été mis en pause.
Et enfin la fonction répète tout cela en boucle jusqu’à la fermeture du debugger.

Le cycle de vie

Le cycle de vie du debugger n'est pas affecté par celui du debuggee. Il ne quittera que par une demande explicite. En revanche, pour être bien sûr de quitter proprement, le debugger devra toujours fermer le debuggee qui lui est associé, s'il en a un, avant de lui-même quitter.
Nous avons fait en sorte de rendre possible le redémarrage d'un debuggee, avec une confirmation auprès de l'utilisateur si un debuggee est déjà en cours. Pour cela il a juste fallu faire attention à fermer l'ancien et à ouvrir le nouveau lors de deux tours de boucle différents, pour que le debugger reçoive les évènements correctement.
Notre debugger connaît trois principaux états : l'attente d'un évènement du debuggee, l'interaction utilisateur et la gestion d'évènement. C'est uniquement

dans ce premier état que le debuggee tourne. Il suffit donc de quitter l'interaction utilisateur pour que, par le biais de la fonction 'ContinueDebugEvent', l'exécution reprenne.

Breakpoint

Un breakpoint n'est rien d'autre qu'une instruction particulière qui va lever une exception de type INT 3. Cette instruction a pour optcode 0xCC. Cette instruction ne comprenant qu'un octet, il est aisé de remplacer le premier de l'instruction à l'adresse mémoire où l'on veut poser un breakpoint dans le debuggee. Ainsi, au moment de lire cette instruction, nous recevront une exception INT 3. On peut donc, à ce moment, interagir avec l'utilisateur alors que le debuggee est mis en pause.

Au moment de relancer le programme, il faudra replacer l'instruction originale (que nous avons donc stockée dans une map adresses / instructions) afin de l'exécuter. Mais puisque le programme a avancé à l'instruction d'après notre 0xCC, il faudra aussi faire reculer EIP d'un octet. La dernière contrainte a été de pouvoir conserver les breakpoints qui ne seront jamais ré-exécutés si l'on ne fait rien de plus. Afin de pouvoir remettre notre breakpoint après avoir exécuté l'instruction originale, nous avons fait appel à une autre exception, l'exception single step. C'est une exception levée lorsque le bit trap flag du registre EFLAGS est mis a 1, à la prochaine instruction exécutée. Ainsi nous pouvons lever puis capturer cette exception afin d'être, comme on le voulait, juste après l'exécution de l'instruction originale. Nous pouvons alors ré-écrire notre breakpoint qui pourra à nouveau déclencher une exception si on l'atteint.

Pour supprimer un breakpoint de notre debuggee, il nous suffit de réécrire à la bonne adresse l'instruction correspondante contenue dans notre map, et de supprimer cet élément de la map. L'affichage des breakpoints actifs peut donc se faire avec un simple parcours de la map.

Nous avons en revanche rencontrée une difficulté : tant que l'utilisateur pose un breakpoint à une adresse valide, il n'y a pas de problème. En revanche si le breakpoint est posé ailleurs que sur le premier bit d'une instruction, il ne déclenchera jamais d'exception et perturbera le déroulement du debuggee. Pour éviter ça, il faut donc pouvoir vérifier que l'adresse choisie est bien celle d'une instruction. Nous avons d'abord pensé à utiliser « Capstone » afin de désassembler le code du debuggee et de vérifier qu'il y avait bien une suite d'instructions valides depuis l'adresse contenue par EIP jusqu'à l'adresse fournie par l'utilisateur (ou l'inverse). Le problème est que « Capstone » ne peut lire que des instructions, et qu'il est tout à fait possible qu'entre deux instructions d'un programme se trouve stockées des données. Il existe bien un mode 'skipdata', mais il nous a paru très approximatif, sautant tout ce qu'il ne reconnaît pas et réinterprétant le code en instructions dès qu'il le peut. Avec cette méthode, il y avait de grandes chances que l'utilisateur ne puisse plus poser de breakpoint à certaines adresses valides (et puisse quand même en poser à de mauvaises). Nous avons donc uniquement implémenté une vérification on ne peut plus basique : tester si les données contenues à l'adresse fournie sont comprises comme instruction par « Capstone ». Cette vérification est très sommaire et empêche peu de mauvais adressages, mais dans l'attente de mieux, nous avons utilisé cette méthode.

Le premier breakpoint :
Lorsque qu'un programme est chargé à l'intérieur d'un débugger, l'OS va automatiquement lever un premier breakpoint. Nous exploitons ce breakpoint pour nous rendre jusqu'au démarrage du debuggee afin de laisser la possibilité à l'utilisateur d'interagir avec son programme en ayant un visuel sur son état.

Steps

Step into :
Le step into consiste à tracer l'exécution du debuggee instruction par instruction. La plupart des solutions que nous avons consultées proposaient d'utiliser l'exception single step pour poser un breakpoint à utilisation unique (qui ne sera pas ré-écrit dans le code une fois atteint), à l'adresse contenue par EIP. Nous avons trouvé plus simple d'utiliser directement uniquement les exceptions single step pour se déplacer. En utilisant un compteur, il nous a été facile de faire en sorte que cette exception se réactive le nombre de fois désiré, si l'utilisateur veux se déplacer de plusieurs instructions d'un coup.

Step over :
Le step over consiste à exécuter le debuggee ligne par ligne et non plus instruction par instruction. C'est à dire que si le programme s'apprête à rentrer dans une sous-routine, nous ne mettrons en pause son exécution qu'une fois qu'il sera ressorti de cette sous-routine. Pour réussir à faire cela, nous avons désassemblé (à l'aide de « Capstone ») l'instruction sur laquelle pointe EIP. Si cette instruction est reconnue comme un 'call', nous posons un breakpoint à utilisation unique, à l'adresse de l'instruction suivante (dont nous obtenons l'adresse là aussi grâce au désassembleur). En revanche, si l'instruction n'est pas un 'call', nous effectuons tout simplement un step into puisqu'il n'y a pas de différence entre les deux steps dans cette situation.

SIGINT

Nous avons voulu faire en sorte de pouvoir mettre en pause le debuggee lorsqu'il est cours d'exécution sans devoir atteindre un breakpoint. Pour cela, nous avons capturé le signal SIGINT qui se produit lorsque l'utilisateur fait un CTLR-C. La capture de ce signal permet donc de redonner la main à l'utilisateur.

Consultations des données

Afin de pouvoir fournir à l'utilisateur des informations sur l'état du debuggee, notre debugger doit être en mesure de les consulter. Aussi, notre debugger est capable de lire le contenu de la mémoire de son debuggee sous la forme de byte, word ou dword en fonction de la taille demandée.

Nous avons également la possibilité de récupérer la structure CONTEXT qui permet, de par ses champs, d'accéder à tous les registres du processeur. Nous avons donc fait en sorte d'afficher le contenu des registres du debuggee avec le détail des bits importants du registre EFLAGS puisque, pour celui-ci, seules comptent leurs valeurs individuelles.