GenDbg : un débogueur générique

Didier Eymery, Odile Eymery, Jean-Marie Borello, Jean-Marie Fraygefond, and Philippe Bion

CELAR, BP 7419, 35174 Bruz Cedex
jean-marie.fraygefond@dga.defense.gouv.fr

Résumé L'objet de cet article est de présenter l'architecture technique d'un outil de débogage générique développé au CELAR[1]. GenDbg est un framework[2] permettant de déboguer tout type d'application sur différentes architectures. Il fusionne à travers une interface unifiée les fonctionnalités de plusieurs outils.

1 Introduction

Le débogueur est l'outil incontournable lors de la phase de mise au point d'un programme ou lors d'analyses «non coopératives» de systèmes. Chaque environnement de développement logiciel intègre un outil de débogage plus ou moins évolué et, devant la multitude des architectures matérielles et des langages, le nombre de ces outils dédiés ne cesse d'évoluer. Certaines solutions techniques se sont révélées non pérènnes, d'autres inadaptées aux besoins ; les outils fournis par les éditeurs ne proposent pas toujours les fonctionnalités requises pour des investigations poussées.

Afin de maintenir notre niveau d'expertise il nous a semblé nécessaire de faire converger et de synthétiser ces outils sous la forme d'une solution générique cohérente et évolutive.

GenDbg est un framework développé à cet effet. GenDbg permet de déboguer tout type de programme (code machine, byte code java, P-Code visual basic, DotNet,...) sur des architectures hétérogènes. Ce projet a été développé au sein du département d'Analyse de la Menace Informatique du CELAR, en réponse à nos besoins spécifiques d'analyse de code.

2 Génèse

2.1 Besoins

Le projet est motivé par trois grands objectifs :

- **Maîtrise**, la maîtrise de l'outil passe par le contrôle de tous les aspects du cycle de vie logiciel, des spécifications au maintien en conditions opération-nelles afin de s'affranchir de toute dépendance vis-à-vis d'un éditeur de logiciel.
  Cette situation s'est déjà présentée avec l'arrêt du débogueur Softice en juillet 2006.
  Cette maîtrise permet d'ajouter des fonctionnalités et de corriger les bugs rencontrés très rapidement.
- **Modularité**, contrairement aux outils monolithiques, la modularité propose un outil constitué d'un coeur et d'une multitude de modules. Cette conception modulaire présente un double avantage : d'une part elle permet un enrichissement fonctionnel très rapide et facilement paramétrable et d'autre part elle permet une bonne répartition de la charge de travail liée au projet.

[1] Centre d'électronique de l'armement
[2] Espace de travail modulaire

– **Généricité**, la généricité représente certainement le principal intérêt du projet, l'objectif étant de permettre l'analyse de n'importe quel programme sur une quelconque architecture. Il doit être possible de «tracer» c'est à dire exécuter pas-à-pas un programme (binaire ou interprété) quel que soit son mode d'exécution (Ring0, Ring3, Superviseur,...) sur n'importe quelle plateforme et n'importe quel système d'exploitation.

2.2 Les principaux outils existants

Nous citons dans cette partie uniquement les outils les plus répandus pour l'analyse dynamique de code. Chaque outil a ses adeptes, ses points forts et ses points faibles. Nous partons du principe qu'il n'y a pas de «bons» ou de «mauvais» outils (quoique) mais des outils qui répondent (ou pas) aux besoins.

– **Softice**, produit par la société Compuware et fonctionnant exclusivement sous Windows, ce débogueur noyau a longtemps été le débogueur de choix pour les systèmes Microsoft. Malheureusement cet outil n'est plus maintenu depuis 2006. Parmis les points forts de Softice on peut citer : une forte intégration dans l'OS permettant de déboguer tout type de composant (applicatifs, services, drivers), une interface mode texte très pratique. Les qualités de Softice sont aussi à la source de ses défauts, à savoir que son intégration en profondeur dans le système n'est pas sans poser des problèmes d'installation et de stabilité, enfin Softice souffre de l'absence d'interfaces de programmation documentées et d'un moteur de script rendant très difficile son enrichissement fonctionnel. Il existe cependant une extension non officielle nommée IceExt [1] rajoutant à softice des fonctionnalités qui lui faisaient cruellement défaut (dump mémoire, copie d'écrans,...).
– **WinDbg**, produit gratuit fournit par Microsoft via les «debugging tools»[2]. C'est le débogueur officiel de plus en plus utilisé depuis l'arrêt de Softice. WinDbg permet de déboguer n'importe quelle application sur une machine distante (physique ou virtuelle) via une liaison série. WinDbg bénéficie d'une très bonne richesse fonctionnelle et d'une très bonne intégration dans l'OS (et pour cause !). Cependant WinDbg est spécifique aux OS Microsoft.
– **Gdb**, le débogueur GNU [3] est un débogueur qui fonctionne sur de nombreux environnements UNIX et supporte plusieurs architectures. Gdb permet de déboguer une machine distante via une liaison série ou TCP/IP. Gdb introduit la notion de «stub» distant. Les stubs Gdb sont en fait des gestionnaires d'exceptions (points d'arrêts, mode trace) prenant en charge un jeu de commandes restreint (lecture/écriture registre & mémoire) véhiculé sur un protocole de communication.
– **OllyDbg** [4], il s'agit d'un débogueur shareware utilisable gratuitement qui permet le débogage applicatif en environnement Win32. Il ne supporte pas le mode noyau (Ring0). OllyDbg bénéficie d'un certain engouement et d'une communauté très active qui développe scripts et plugins pour ce projet. OllyDbg est cependant très limité en terme de cibles.
– **Syser** [5], il s'agit d'une initiative Chinoise assez récente pour développer un successeur de Softice, à savoir un debogueur Ring0 pour les OS Windows.

Ce panorama des outils de débogage est loin d'être exhaustif, on pourrait encore citer ImmunityDebugger [6] ou le Rasta Ring0 debugger [7] qui fait un premier pas vers la généricité en étant indépendant de l'OS (Windows, Linux, BSD), malheureusement il n'a pas fait le deuxième pas vers l'indépendance vis à vis de l'architecture puisqu'il est spécifique aux processeurs Intel x86, ...

Il apparait un certain nombre de limitations communes : force est de constater qu'aucun outil n'est en mesure de déboguer une application en Ring0 à la fois sous Unix et sous Windows, aucun

outil ne sait tracer du P-Code, aucun outil ne sait gérer des architectures avec des tailles de cellules differentes de 8 bits,...

Ce qui nous conduit à une tentative d'élaboration d'un outil générique.

2.3 Choix techniques retenus

Après avoir manipulé la plupart des outils précédents et forts des constats énoncés, nous avons pu dégager les principes de base d'un débogueur générique : GenDbg.

- **IHM mode texte**, «à la Softice», pour des raisons d'habitude principalement et parce que nous considérons que cette IHM a fait ses preuves, plus rapide et plus puissante que les menus déroulants et autres «clickodromes». Un aperçu de l'interface GenDbg est visible sur la figure 1. L'espace d'affichage est découpé en zones typées : code (WC), data (WD), registres (WR).
- **Modèle Framework\Stubs**, car ce modèle permet de séparer les composants débogués et la machine de l'analyste. En fonction du type d'analyse envisagé (malware) on préfèrera travailler sur deux machines physiques distinctes, les machines virtuelles actuelles (virtual PC et WMware) étant loin d'être «transparentes». Cette approche Stub permet aussi de réduire l'empreinte du débogueur sur la cible.
- Enfin, cette approche framework\stubs va simplifier le développement en s'affranchissant de l'écriture de drivers video et clavier spécifiques à la cible et va rendre le framework et les modules complètement indépendants de l'architecture cible, le stub prenant en charge les aspects spécifiques de la gestion des exceptions bas niveaux sur le processeur cible (x86, 68000, ARM,...).
- **Architecture ouverte et modulaire**, pour permettre le développement de modules tiers et garantir l'évolutivité de l'outil.
- Le framework et les modules sont développés en langage C dans l'environnement Win32 et implémentés sous la forme d'exécutables au format PE (exe et dll). Les stubs sont quant à eux très dépendants de la cible et sont programmés en fonction de la cible (C, Assembleur,...).

L'architecture produite est décrite dans la suite de ce document.

3 Architecture

GenDbg adopte une architecture modulaire ; on distingue deux catégories de modules : les modules de base, obligatoires, qui forment l'architecture «minimale» présentée en figure 2 et les modules optionnels qui complètent l'architecture minimale pour donner l'architecture «complète» mono-stub présentée en figure 4.

Les modules obligatoires sont :

- **Framework**
- Module de désassemblage (**AsmModule**)
- Stub

Les modules optionnels sont :

- Module de commandes additionnelles (**CmdModule)**
- Module de gestion de symboles (**SymModule)**
- Module OS (**OSHelperModule**)
- Module de gestion de points d'arrêts additionnels **(BpModule)**

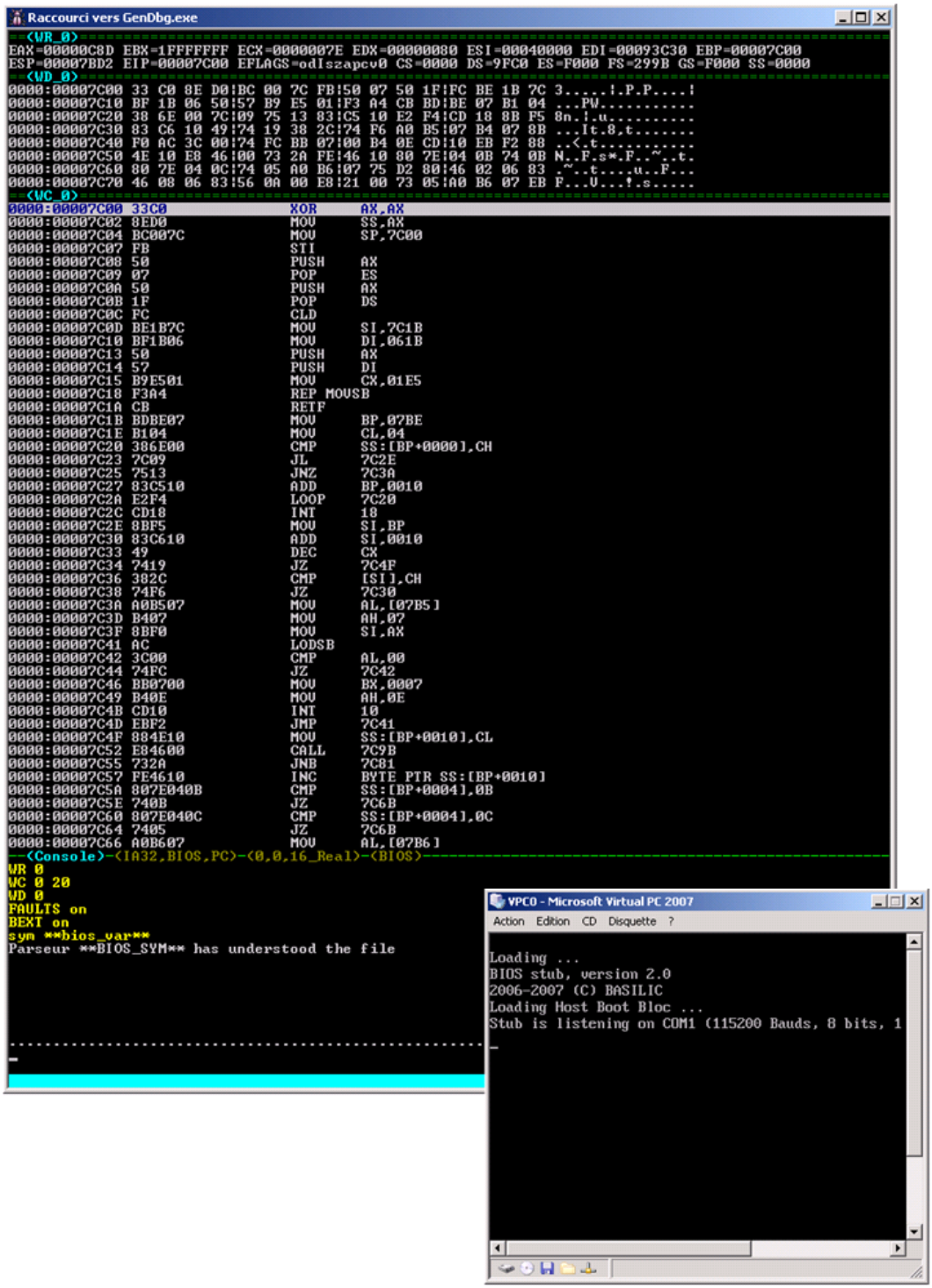

FIG. 1: GenDbg connecté sur le stub Bios

Les rôles et les interfaces de programmation de tous ces modules seront détaillés dans les sections suivantes.

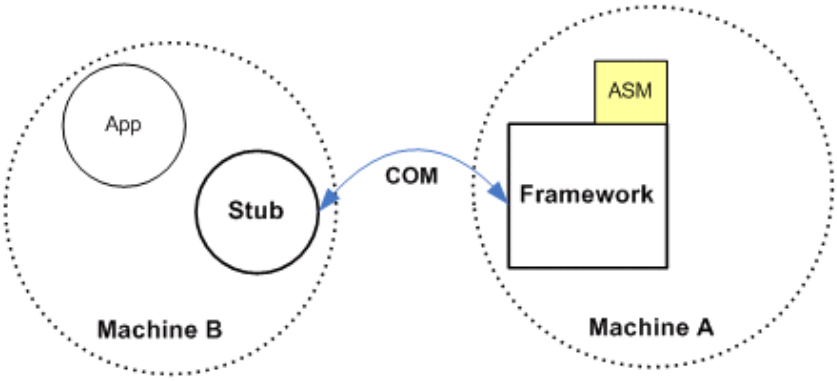

FIG. 2: Architecture minimale

3.1 Structures de données

Afin de parvenir à la généricité il convient de définir quelques abstractions logicielles «objets». Les objets manipulés par un débogueur peuvent se résumer aux structures de données suivantes :

Addresse mémoire : une adresse mémoire est un objet dont la structure est la suivante :

```
typedef struct {
DWORD AddressSpace;
DWORD AddressType;
DWORD szAddress;
BYTE Address[SZ_MAX_ADDRESS];
} MemoryAddress_T;
```

La représentation du champ Address est complètement dépendante de l'architecture et doit être définie par le module de désassemblage qui peut gérer plusieurs types d'adresses (ex : linéaire, virtuelle, physique,...) dans plusieurs espaces d'adressages (il n'y a pas que la RAM !!). Pour chaque architecture supportée on définira un fichier include (.h) qui contiendra l'énumération des différents types d'adresses et la définition de leurs structures internes.

Exemple pour l'architecture IA32, dans ia32.h on définit deux espaces d'adressage et quatre types d'adresses mémoires :

```
// Address definitions
typedef enum {
```

```
IA32_Memory = 0,
IA32_IO
} IA32_AddressSpace_T;

typedef enum {
IA32_ProtectedVirtual = 0,
IA32_Linear,
IA32_Physical,
IA32_RealVirtual
} IA32_AddressType_T;

#pragma pack(1)
typedef struct
{
ULONGLONG Offset;
WORD Segment;
} IA32_ProtectedVirtualAddress_T;
#pragma pack()

#pragma pack(1)
typedef struct
{
ULONGLONG Address;
} IA32_PhysicalAddress_T;
#pragma pack()
```

...

Zone mémoire : une zone mémoire est un objet défini par une adresse de départ et un nombre de cellules, la cellule étant l'unité élémentaire d'adressage mémoire (8 bits ou autre) sur l'architecture cible.

```
typedef struct {
MemoryAddress_T Address;
DWORD NbCell;
} MemoryArea_T;
```

Registre : représente un registre interne d'un microprocesseur ; on identifie un registre avec un triplet constitué par : le CPU concerné (architectures multiprocesseurs ou multicoeurs), la banque (exemple : les processeurs ARM ont une banque de registres par mode fonctionnement), et le nom/n° du registre.

```
typedef struct {
DWORD NoCPU;
DWORD NoBank;
DWORD RegisterId;
} CPURegisterIdent_T;
```

Contexte CPU : le contexte CPU représente l'état courant d'un microprocesseur. Il s'agit d'une structure propre à l'architecture cible.

Les champs NoCPU et NoBank du contexte CPU représentent le CPU et la Banque de registres actifs au moment la prise de vue (Break).

```
typedef struct {
DWORD NoCPU;
DWORD NoBank;
DWORD sz;
BYTE Ctx[SZ_MAX_CPU_CTX];
} CPUCtx_T;
```

Contexte OS : le champ Ctx sert à la représentation interne du contexte de l'OS. La représentation interne de ce contexte est définie par le module OSHelper associé au système d'exploitation cible.

```
typedef struct {
DWORD sz;
BYTE Ctx[SZ_MAX_OS_CTX];
} OSCtx_T;
```

A titre d'exemple, le contexte d'un OS Windows NT peut comprendre : le ProcessID, le ThreadID, la valeur du registre CR3 (adresse de la table des pages du processus).

Vue : ViewCtx_T fournit des informations contextuelles propres à la vue courante dans le framework. Cette structure réunit le CPUCtx et l'OSCtx. Elle contient en plus une information IsBreakCtx utilisé pour gérer le filtrage (voir section filtrage).

```
typedef struct {
BOOL IsBreakCtx;
CPUCtx_T CPUCtx;
OSCtx_T OSCtx;
} ViewCtx_T;
```

3.2 Modules de base

Dans cette section nous décrivons un peu plus en détail les fonctionnalités de chaque type de module ainsi que les mécanismes de communication intermodules.

Framework : le framework représente le «coeur» de l'outil. C'est une application Win32 qui prend en charge la communication inter-modules, l'interface homme-machine, implémente le jeu de commandes de base commun à l'ensemble des modules ainsi que le moteur de script.

Tous les modules sont implémentés sous forme de librairies dynamiques et sont chargés dans l'espace d'adressage du framework après identification de la cible par un stub.

Le framework a en charge la lecture d'un fichier de configuration (.ini) pour tenter d'établir un canal de communication avec un stub et l'initialisation des différents modules. Dans l'état actuel du projet le framework est capable de communiquer avec les stubs via un port série ou des canaux

nommés (pipes). Il est ainsi possible de déboguer aussi bien des machines physiques que des machines virtuelles (VMWare ou VirtualPC) en mappant un port série virtuel sur un canal nommé.

Le framework fournit aux différents modules un jeu de fonctions de base qui vont permettre d'intéragir entre eux.

Ci-dessous la liste des «helper» fonctions proposées par le framework pour des modules de commandes additionnelles. Notez que le framework n'implémente pas la majorité de ces fonctions. Le framework joue le rôle d'intermédiaire et d'aiguilleur entre les modules.

```
typedef enum {
GenDbgHelperCmd_ReadRegister = 0,
GenDbgHelperCmd_ReadMemory,
GenDbgHelperCmd_WriteConsoleWnd,
GenDbgHelperCmd_AddressToTxt,
GenDbgHelperCmd_TxtToAddress,
GenDbgHelperCmd_DataToTxt,
GenDbgHelperCmd_TxtToData,
GenDbgHelperCmd_GetAddressInfo,
GenDbgHelperCmd_GetNearestAddressInfo,
GenDbgHelperCmd_IsCellAreaValid,
GenDbgHelperCmd_WriteRegister,
GenDbgHelperCmd_WriteMemory,
GenDbgHelperCmd_Private,
GenDbgHelperCmd_AddScriptVar,
GenDbgHelperCmd_Refresh,
GenDbgHelperCmd_LastFnIdx
} GenDbgHelperCmdFnIdx_T;
```

- Un module de commande peut demander à un stub de lire de la mémoire (GenDbgHelperCmd_ReadMemory)
- Un module de commande peut demander à un OSHelper des informations sur une Adresse mémoire (GetAddressInfo)
- Un module de commande peut demander au framework un affichage (GenDbgHelperCmd_WriteConsoleWnd)
- ...

Stubs : les stubs s'exécutent sur la machine cible et prennent en charge la gestion des exceptions sur l'architecture cible ; à ce titre ils sont fortement dépendants de l'architecture et de l'OS de la cible. Les stubs fournissent les services de base au framework, c'est-à-dire les accès en lecture et en écriture aux registres et à la mémoire. Ils réalisent aussi l'identification précise de la cible pour le framework, l'identification conduisant au chargement des modules utiles et disponibles. Les stubs communiquent avec le framework à l'aide d'un protocole décrit un peu plus loin dans cet article.

ASM Modules : les ASM modules sont en fait des désassembleurs/assembleurs prenant en charge la représentation du code, des données et des adresses propres à l'architecture cible. Par exemple sur l'architecture IA32, en mode virtuel protégé une adresse mémoire est de la forme «sélecteur :offset» avec le sélecteur sur 16 bits et l'offset sur 32 bits. C'est le module de désassemblage IA32 qui a la charge de la représentation visuelle d'une adresse virtuelle.

Un module d'assemblage/desassemblage doit exporter la fonction «GetAsmModuleInfo» dont le prototype est le suivant :

```
typedef BOOL (__cdecl* GetAsmModuleInfoFn_T)(AsmModuleInfo_T** AsmModuleInfo);
```

Le type AsmModuleInfo_T est décrit ci-dessous :

```
typedef struct {
DWORD Version;
char* Architecture; // Ex : IA32
DWORD szCellInBits;
DWORD szMaxInstructionInCell;
AsmDataInfo_T* TblDataType;
AsmGroupRegisterInfo_T* TblGroupRegister;
AsmAddressTypeInfo_T* TblAddressType;
AsmAddressSpaceInfo_T* TblAddressSpace;
AsmBankInfo_T* TblBank;
char* CPUCtxSyntaxHelp;
char* AddressSyntaxHelp;
char* DataSyntaxHelp;
char* ConditionSyntaxHelp;
void* FunctionTbl[AsmModule_LastFnIdx];
} AsmModuleInfo_T;
```

Cette structure de données va permettre de décrire les éléments caractéristiques de l'architecture supportée. Parmi les données intéressantes on notera :

szCellInBits : taille d'une cellule mémoire en bits, GenDbg peut gérer des tailles de cellules différentes de 8 bits !
TblDataType : définit les types de données de l'architecture cible (WORD, DWORD, IA32_DESCRIPTOR, CPUID, ...)
TblGroupRegister : définit des groupes de registres (généraux, MMX, SSE2, DBG,...)
TblAddressSpace : définit les plans mémoires adressables.
TblBank : informations concernant les banques de registres.
FunctionTbl fournit une table de fonctions "exportées" par le module ASM ; une API avec laquelle le framework et les autres modules vont pouvoir manipuler les objets tels que les Adresses mémoires, les contextes et les types de données.

Exemple d'initialisation pour le module ASM prenant en charge l'architecture IA32 :

```
AsmModuleInfo_T GenericAsmModuleInfo={ 1,//Version
"IA32",//Architecture
IA32_SZ_CELL,//szCellInBits
IA32_SZ_MAX_INSTRUCTION,
TblDataType,
TblGroupRegister,
TblAddressType,
TblAddressSpace,
NULL,
```

```
"16/32 Bits\n",

"Addressing mode\nSegment:Offset as default\n
                    # for physical\n$ for linear\n
                    & for switching (Real/Protected)\n",

"Empty\n",
"Condition Synthax as C one\n",
{&Init,
      &Uninit,
      &GetLastErrorMsg,
      &AssembleSingle,
      &UnassembleBloc,
      &DataToTxt,
      &TxtToData,
      &AddressToTxt,
      &TxtToAddress,
      &GetInstructionInfo,
      &IsConditionSatisfied,
&IsAddressInMemoryArea,
&CompareAddress,
&EvalAddress,
&EvalOffset,
&TxtToCPUCtx,
&CPUCtxToTxt,
&GetInstructionPointer,
&ReadValueFromThisValue}
};
```

L'implémentation de la fonction GetAsmModuleInfo est triviale :

```
BOOL __cdecl GetAsmModuleInfo(AsmModuleInfo_T** AsmModuleInfo)
{
if (AsmModuleInfo==NULL) return FALSE;
(*AsmModuleInfo)=&GenericAsmModuleInfo;
return TRUE;
}
```

Une dernière remarque concernant les modules ASM, il n'est pas prévu d'avoir plusieurs modules ASM pour une même architecture (pour avoir des représentations de l'assembleur différentes syntaxiquement par exemple AT&T/Intel).

3.3 Processus d'initialisation d'un module

Avant d'aborder la description des modules additionnels il convient de présenter les aspects dynamiques de l'initialisation de l'architecture minimale.

L'initialisation des différents modules (ASM, CMD, OS_HELPERS, SYM, BP) par le framework obeit au processus décrit figure 3.

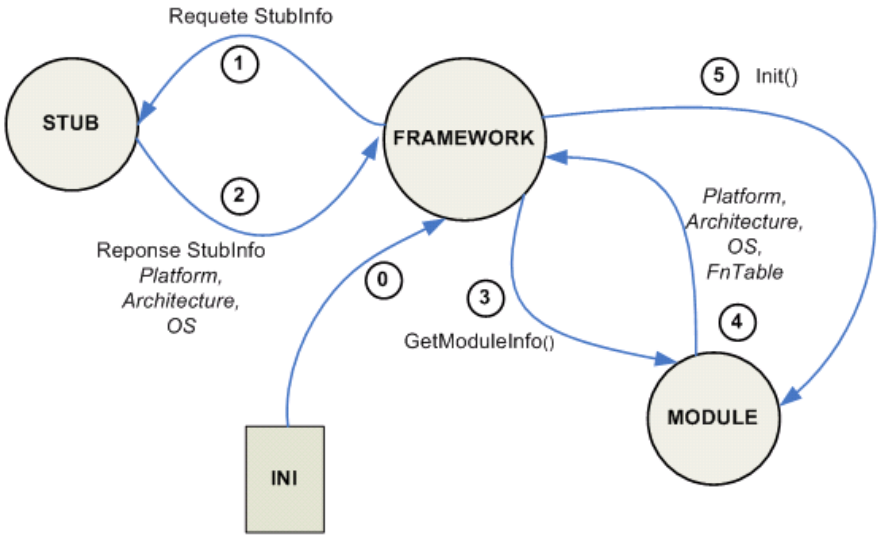

FIG. 3: Processus d'initialisation de module

- 0 - Le framework lit son fichier de configuration et tente d'établir une communication avec le stub déclaré.
- 1 - Le framework émet une requête de type StubInfo pour recueillir des informations sur la cible.
- 2 - Le stub retourne les informations d'identification (Platform/Architecture/OS)
- 3 - Le framework charge les modules disponibles et appelle GetModuleInfo()
- 4 - Chaque module répond en retournant une structure de données spécifique. Le framework filtre les modules compatibles grâce aux informations d'identification fournies.
- 5 - S'il y a compatibilité, le framework enregistre la table des fonctions fournie par le module et lui procure en retour la liste des helpers fonctions dont il peut avoir besoin via un appel à sa fonction Init() en fournissant au module un pointeur sur une structure de type GenDbgHelperCmdInfo_T.

Ci-dessous l'implémentation de la fonction Init() dans un module de commande.

```
BOOL __cdecl CmdModuleGenIA32_Init(GenDbgHelperCmdInfo_T* HelperInfos,
void** hModuleInstance)
{
CmdModuleGenIA32Ctx_T* CmdModuleGenIA32Ctx; //une structure privée du module CMD
CmdModuleGenIA32Ctx = malloc(sizeof(CmdModuleGenIA32Ctx_T));
CmdModuleGenIA32Ctx->GenDbgHelperCmdInfo = HelperInfos;
CmdModuleGenIA32Ctx->LastError = CmdModuleErrorNoError;
(*hModuleInstance) = CmdModuleGenIA32Ctx;
return TRUE;
}
```

3.4 Modules additionnels

En plus des modules obligatoires, un certain nombre de modules spécifiques optionnels ont été définis afin de faciliter le travail de l'analyste.

Command Modules , les modules de commandes complètent les commandes de base implémentées dans le framework avec des commandes spécifiques au contexte d'exécution de la cible.

Par exemple si la cible est identifiée comme IA32 le framework chargera tous les modules de commande prenant en charge cette architecture. Nous avons développé un module de commande GenericIA32 implémentant des commandes telles que IDT, GDT, LDT permettant d'afficher les tables de descripteurs systèmes propres à l'architecture IA32.

Les structures de données spécifiques aux CmdModules sont données ci-après :

```
typedef enum {
Cmd_ProcessCmd = 0,
Cmd_LastFnIdx
} CmdFnIdx_T;

typedef enum {
CmdModule_Init = 0,
CmdModule_Uninit,
CmdModule_GetLastErrorMsg,
CmdModule_LastFnIdx
} CmdModuleFnIdx_T;

struct CmdModuleInfo_T_;
typedef struct CmdInfoElt_T_ {
struct CmdInfoElt_T_* PrevElt;
struct CmdInfoElt_T_* NextElt;
struct CmdModuleInfo_T_* CmdModuleInfo;
char* CmdName; // MUST BE : en majuscule
char* CmdDescription; // Afficher au listing des commandes
char* CmdMemo; // Afficher au moment de la saisie de la commande
char* CmdHelp; // aide complete
void* FunctionTbl[Cmd_LastFnIdx];
} CmdInfoElt_T;

typedef struct CmdModuleInfo_T_ {
DWORD Version;
char* Plateform; // Ex : PC
char* Architecture; // Ex : IA32 ==> designe le module ASM
char* OS; // Ex : NT
CmdInfoElt_T* ListCmdInfo;
void* FunctionTbl[CmdModule_LastFnIdx];
} CmdModuleInfo_T;

typedef BOOL (__cdecl* GetCmdModuleInfoFn_T)(CmdModuleInfo_T** CmdModuleInfo);
```

Un module de commande exporte une fonction GetCmdModuleInfo() qui fournit au framework un pointeur sur une structure de type CmdModuleInfo_T. Cette structure de données renseigne le framework sur les cibles supportées par le module et fournit les pointeurs de fonctions associées aux commandes implémentées dans le module (via la liste chainée ListCmdInfo).

OS Helpers Modules : les OSHelpers offrent des services et des fonctionnalités propres à un système d'exploitation parmi lesquelles la signalisation d'un certain nombres d'événements internes : chargement/déchargement d'un module (au sens OS du terme), création/destruction de processus ou encore l'appartenance d'une adresse à un module.

Un module OSHelper agit en tant que filtre du stub auquel il est rattaché ; cet aspect est détaillé dans la section filtrage.

Le module exporte une fonction GetOSHelperInfo(...). Cette fonction fournit au framework un pointeur sur une structure de données de type OSHelperInfo_T. Les valeurs des champs Platform/Architecture/OS permettent au framework de déterminer la compatibilité de ce module avec le stub courant.

```
typedef enum {
OSHelper_Init = 0,
OSHelper_Uninit,
OSHelper_GetLastErrorMsg,
OSHelper_GetAddressInfo,
OSHelper_GetModuleBaseName,
OSHelper_ReadRegister,
OSHelper_WriteRegister,
OSHelper_ReadMemory,
OSHelper_WriteMemory,
OSHelper_TxtToOSCtx,
OSHelper_OSCtxToTxt,
OSHelper_GetViewOSCtx,
OSHelper_CompareOSCtx,
OSHelper_Private,
OSHelper_SetBreakpoint,
OSHelper_UnsetBreakpoint,
OSHelper_LastFnIdx
} OSHelperFnIdx_T;

typedef struct {
DWORD Version;
char* Plateform; // Ex : PC
char* Architecture; // Ex : IA32 ==> designe le module ASM
char* OS; // Ex : NT
char* OSCtxSyntaxHelp;
void* FunctionTbl[OSHelper_LastFnIdx];
} OSHelperInfo_T;

typedef BOOL (__cdecl* GetOSHelperInfoFn_T)(OSHelperInfo_T** OSHelperInfo);
```

Breakpoint Helpers Modules : les modules de points d'arrêts additionnels permettent la prise en charge et la définition de points d'arrêts spécifiques à l'architecture cible. Par exemple la gestion des BPM (hardware breakpoints) Intel sont pris en charge via un module additionnel.

Les modules de gestion de points d'arrêts additionnels s'apparentent à des modules de commandes.

```
// Un module de BreakPoint doit exporter la fonction : GetBreakPointModuleInfo
typedef enum {
BreakPoint_Set = 0,
BreakPoint_Enable,
BreakPoint_Disable,
BreakPoint_Clear,
BreakPoint_View,
BreakPoint_IsThisAddressABreakAddress,
BreakPoint_LastFnIdx
} BreakPointFnIdx_T;

typedef enum {
BreakPointModule_Init = 0,
BreakPointModule_Uninit,
BreakPointModule_GetLastErrorMsg,
BreakPointModule_LastFnIdx
} BreakPointModuleFnIdx_T;

struct BreakPointModuleInfo_T_;
typedef struct BreakPointExInfoElt_T_ {
struct BreakPointExInfoElt_T_* PrevElt;
struct BreakPointExInfoElt_T_* NextElt;
struct BreakPointModuleInfo_T_* BreakPointModuleInfo;
char* BreakPointName; // MUST BE : en majuscule (Should begin with a Bxxx)
char* BreakPointDescription; // Afficher au listing des commandes
char* BreakPointMemo; // Afficher au moment de la saisie de la commande
char* BreakPointHelp; // aide complete
void* FunctionTbl[BreakPoint_LastFnIdx];
} BreakPointExInfoElt_T;

typedef struct BreakPointModuleInfo_T_ {
DWORD Version;
char* Plateform; // Ex : PC
char* Architecture; // Ex : IA32 ==> designe le module ASM
char* OS; // Ex : NT
BreakPointExInfoElt_T* ListBreakPointExInfo;
void* FunctionTbl[BreakPointModule_LastFnIdx];
} BreakPointModuleInfo_T;

typedef BOOL (__cdecl* GetBreakPointModuleInfoFn_T)
(BreakPointModuleInfo_T** BreakPointModuleInfo);
```

Symbol Modules Les modules de symboles permettent d'associer un symbole à une adresse particulière dans un module de l'OS cible.

```
typedef enum {
Symbol_GetSymbolList = 0,
Symbol_FreeSymbolList,
Symbol_LastFnIdx
} SymbolFnIdx_T;

typedef enum {
SymbolModule_Init = 0,
SymbolModule_Uninit,
SymbolModule_GetLastErrorMsg,
SymbolModule_LastFnIdx
} SymbolModuleFnIdx_T;

struct SymbolModuleInfo_T_;
typedef struct SymbolInfoElt_T_ {
struct SymbolInfoElt_T_* PrevElt;
struct SymbolInfoElt_T_* NextElt;
struct SymbolModuleInfo_T_* SymbolModuleInfo;
char* ParseurName;
void* FunctionTbl[Symbol_LastFnIdx];
} SymbolInfoElt_T;

typedef struct SymbolModuleInfo_T_ {
DWORD Version;
SymbolInfoElt_T* ListSymbolInfo;
void* FunctionTbl[SymbolModule_LastFnIdx];
} SymbolModuleInfo_T;

typedef BOOL (__cdecl* GetSymbolModuleInfoFn_T)(SymbolModuleInfo_T** SymbolModuleInfo);

//-----------------------------------------------------
typedef BOOL (__cdecl* SymbolModule_InitFn_T)(void** hModuleInstance);
typedef BOOL (__cdecl* SymbolModule_UninitFn_T)(void* hModuleInstance);

//-----------------------------------------------------
#define SZ_MAX_SYMBOL_NAME 256
typedef struct {
DWORD RVAAddress;
char* Name;
} SymbolDescription_T;

#define SYMBOL_LAST_DESCRIPTION_INFO {0,NULL}

typedef BOOL (__cdecl* Symbol_GetSymbolListFn_T)(void* hModuleInstance,
```

```
char* FilePath,char** ModuleName,SymbolDescription_T** TblSymbol);

typedef BOOL (__cdecl* Symbol_FreeSymbolListFn_T)(void* hModuleInstance,
char* ModuleName,
SymbolDescription_T* TblSymbol);
```

L'architecture complète, avec l'ensemble des modules additionels disponibles est présentée en figure 4.

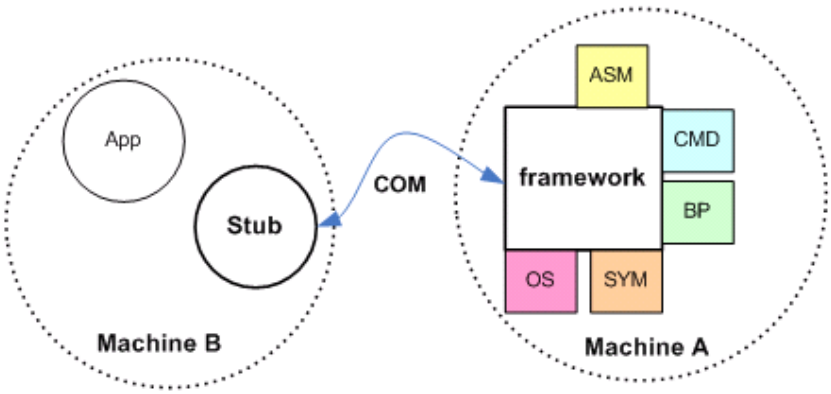

FIG. 4: Architecture complète mono-stub

3.5 Protocole de communication

Le framework communique avec les stubs à l'aide d'un protocole basé sur l'envoi/réception de paquets de données. Le format des paquets est décrit dans la figure 5. Un paquet est constitué d'un en-tête obligatoire et éventuellement de données.

Le protocole est volontairement simple et se définit en quelques lignes.

```
typedef enum { // BigEndian representation
GenDbgCmdType_Requette = 0,
GenDbgCmdType_Reponse,
GenDbgCmdType_Event,
GenDbgCmdType_PrivateRequette,
GenDbgCmdType_PrivateReponse,
GenDbgCmdType_LastCmdType
} GenDbgCmdType_T;

typedef enum { // BigEndian representation
```

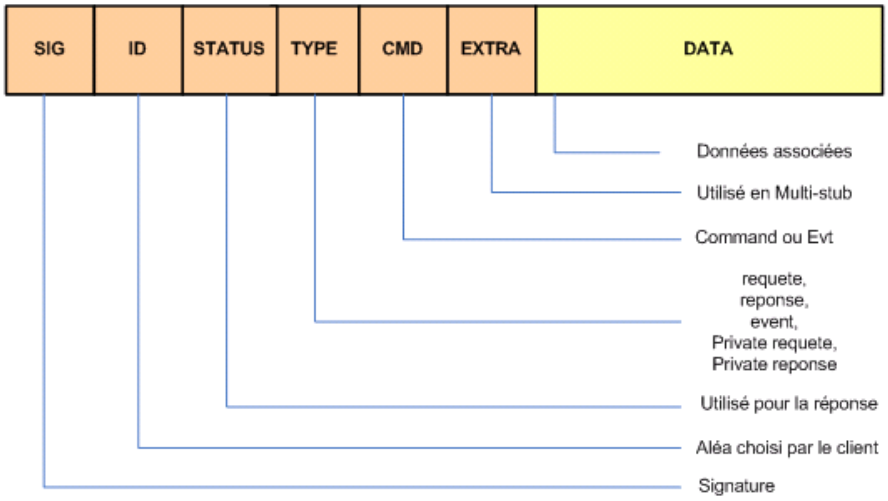

FIG. 5: Paquet GenDbg

```
GenDbgCmd_Ping = 0,
GenDbgCmd_StubInfo,
GenDbgCmd_ReadMemory,
GenDbgCmd_WriteMemory,
GenDbgCmd_ReadRegister,
GenDbgCmd_WriteRegister,
GenDbgCmd_SetBreakpoint,
GenDbgCmd_UnsetBreakpoint,
GenDbgCmd_GetExecCtx,
GenDbgCmd_LastCmd
} GenDbgCmd_T;

typedef enum { // BigEndian representation
GenDbgEvt_BreakEncoutered = 0,
GenDbgEvt_UserManualBreak,
GenDbgEvt_Continue,
GenDbgEvt_Trace,
GenDbgEvt_Ack,
GenDbgEvt_StubLoaded,
GenDbgEvt_StubUnloaded,
GenDbgEvt_KeepAlive,
GenDbgEvt_LastEvt
} GenDbgEvt_T;
```

A noter la capacité pour les stubs d'implémenter des fonctions «privées» pour répondre à des besoins spécifiques des modules. Ex : le stub ring0 windows est implémenté sous la forme d'un driver ; ce stub implémente une fonction privée qui retourne un pointeur sur le DRIVER_OBJECT associé au stub. Le module de commandes additionnelles pour les OS Windows utilise cette fonction privée du stub pour obtenir un point d'entrée sur la liste des modules chargés par l'OS.

3.6 Configuration

La configuration de GenDbg s'effectue avec un fichier .ini. La section [GENDBG] contient les paramètres qui vont permettre l'initialisation de certaines variables internes et surtout le DEBUGER_STUB_PATH qui va indiquer au framework comment initier la communication avec un stub. Dans l'exemple ci-dessous le framework tentera de communiquer avec le stub via un canal nommé com_1

```
[GENDBG]
DEBUGER_STUB_PATH=\\.\PIPE\com_1
;DEBUGER_STUB_PATH=\\.\COM1
```

La section [STARTUP] contient une liste de commandes à exécuter au démarrage.

```
[STARTUP]
CMD0=WC 0 20
CMD1=FAULTS ON
CMD2=BEXT ON
...
```

4 Fonctionnalités avancées

4.1 Multistubs

Un des points forts de GenDbg est la possibilité d'utiliser plusieurs stubs pour déboguer une même application. En effet, la plupart des langages interprétés peuvent faire appel au langage natif de l'architecture sur laquelle l'application tourne. Il est alors intéressant de pouvoir travailler sur les deux langages en même temps. C'est à cet effet qu'a été conçue l'architecture multi-stubs. La figure 4 présente en réalité une vue simplifiée du framework dans le cas de l'utilisation d'un unique stub où le framework se confond avec le StubHandler, c'est-à-dire l'objet en charge d'un stub.

Dans le cas ou plusieurs stubs sont utilisés, chacun est en communication avec un StubHandler qui lui est propre via un unique canal de communication. Plus précisément, chaque sous-stub est rattaché à un MasterStub qui est en charge de la distribution des messages entre sous-stubs et StubHandler. Ainsi, le framework est le contrôleur de tous les StubHandlers. Chaque StubHandler possède ses propres modules chargés en fonction du sous-stub auquel il est rattaché.

Cette architecture complète est détaillée sur la figure 6.

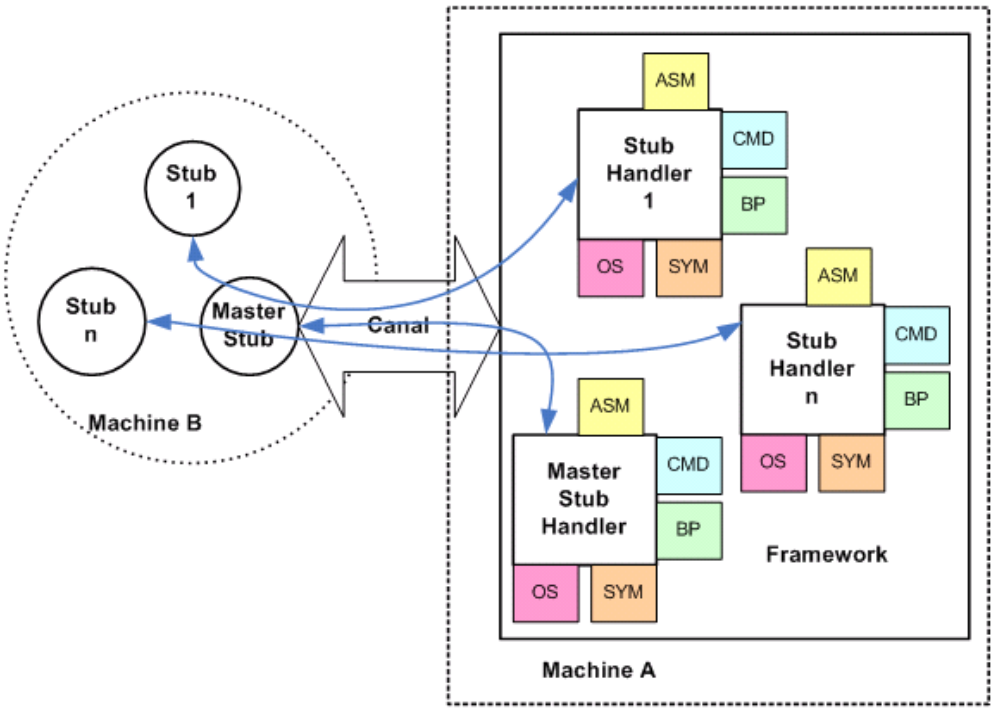

FIG. 6: Architecture complète multi-stubs

4.2 Scripting

L'implémentation d'un moteur de script permet l'automatisation des tâches fastidieuses pour l'analyste. Cette fonctionnalité accroit considérablement les possibilités de GenDbg. Le langage de script implémenté dans GenDbg possède les caractéristiques suivantes :

- langage à substitution proche syntaxiquement du Batch MS-DOS.
- jeu d'instructions réduit mais suffisant (philosophie RISC), pas de boucles (for, do...while,...)
- les commandes ne renvoient pas des flux de données mais des données structurées.
- orienté gestion des données (structures/piles/listes)

Un exemple de script est donné en annexe A.

Il est bien évidemment possible d'écrire toute sorte de scripts tels que : reconstruction d'IAT, Unpacking,...

Toute commande implémentée dans un module de commande est «scriptable». le prototype d'une commande est décrit ci-dessous :

```
typedef BOOL (__cdecl* Cmd_ProcessCmdFn_T)(void* hModuleInstance,
                                            ViewCtx_T* ViewCtx,
                                            DWORD argc,
```

```
                                        char** argv,
                                        ScriptCtx_T* ScriptCtx);
```

lorsque la commande est exécutée dans un script elle reçoit en argument un pointeur sur un contexte de script (ScriptCtx) qui lui permet d'interagir avec ce script pour retourner des données structurées grâce à une fonction du framework GenDbgHelperCmd_AddScriptVar.

```
if(ScriptCtx != NULL) {
AddScriptVarFn(CmdModuleGenIA32Ctx->GenDbgHelperCmdInfo->hDbgInstance,
               ScriptCtx,
               "PageBase",
               SCRIPT_INT_LE_T,
               (BYTE*)&PagingCtx.PG__CR0,
               sizeof(DWORD));
}
```

4.3 Filtrage

Le filtrage est une fonctionnalité propre aux modules OSHelpers. L'objectif du filtrage est d'effectuer un pre/post traitement nécessaire pour exécuter les commandes primitives du stub afin de «traduire» la requête de lecture/écriture de registre ou de mémoire pour l'OSCtx spécifié en entrée.

Ci-dessous la liste des helpers fonctions fournies par le framework au module OSHelper suivie de l'api implémentée par l'OSHelper et des prototypes des fonctions GenDbgHelperOSHelper_RawReadRegister & OSHelper_ReadRegister.

```
typedef enum {
GenDbgHelperOSHelper_RawReadRegister = 0,
GenDbgHelperOSHelper_RawWriteRegister,
GenDbgHelperOSHelper_RawReadMemory,
GenDbgHelperOSHelper_RawWriteMemory,
GenDbgHelperOSHelper_RawSetBreakPoint,
GenDbgHelperOSHelper_RawUnsetBreakPoint,
GenDbgHelperOSHelper_IsCellAreaValid,
GenDbgHelperOSHelper_RawPrivate,
GenDbgHelperOSHelper_RegisterBreakEvent,
GenDbgHelperOSHelper_UnregisterBreakEvent,
GenDbgHelperOSHelper_LastFnIdx
} GenDbgHelperOSHelperFnIdx_T;


typedef enum {
OSHelper_Init = 0,
OSHelper_Uninit,
OSHelper_GetLastErrorMsg,
OSHelper_GetAddressInfo,
OSHelper_GetModuleBaseName,
OSHelper_ReadRegister,
```

```
OSHelper_WriteRegister,
OSHelper_ReadMemory,
OSHelper_WriteMemory,
OSHelper_TxtToOSCtx,
OSHelper_OSCtxToTxt,
OSHelper_GetViewOSCtx,
OSHelper_CompareOSCtx,
OSHelper_Private,
OSHelper_SetBreakpoint,
OSHelper_UnsetBreakpoint,
OSHelper_LastFnIdx
} OSHelperFnIdx_T;

typedef BOOL (__cdecl* GenDbg_RawReadRegisterFn_T)(void* hDbgInstance,
CPURegisterIdent_T* CPURegisterIdent,
BYTE* Data,
DWORD szData);

typedef BOOL (__cdecl* GenDbg_ReadRegisterFn_T)(void* hDbgInstance,
ViewCtx_T* ViewCtx,
CPURegisterIdent_T* CPURegisterIdent,
BYTE* Data,
DWORD szData);
```

On remarquera que la fonction de type GenDbg_ReadRegisterFn_T possède un paramètre supplémentaire : le ViewCtx que n'a pas la fonction de type GenDbg_RawReadRegisterFn_T.

Le ViewCtx contient l'OSCtx qui permet à l'OSHelper de déterminer via le champ IsBreakCtx si l'action demandée se situe dans le contexte courant ou dans un autre contexte, auquel cas l'OSHelper doit probablement réaliser un certain nombre d'actions pour accéder à ce contexte d'exécution.

Par exemple, certains OS (Windows notamment) utilisent le registre CR3 pour stocker l'adresse de la table des pages d'un processus. L'OSHelper peut et doit modifier la valeur du registre CR3 pour aller lire ou écrire dans l'espace d'adressage d'un processus particulier (spécifié dans l'OSCtx).

4.4 Etat d'avancement du projet

Nous présentons ci-dessous la liste des stubs développés ou en cours de développement. Certains stubs sont implémentés nativement d'autres le sont sous la forme de «wrapper» qui réalisent la traduction du protocole GenDbg vers le protocole d'un autre debugger (typiquement Gdb).

stubs

- famille Windows NT/IA32 (Ring0 & Ring3) Le stub Ring3 s'appuie sur la Debug API Microsoft.
  - le stub Ring3 dispose d'un sous-stub Visual Basic (P-Code)
- Linux/IA32 Ring3 (PTrace API)
- BIOS/IA32

- Wrapper Gdb
- Java (en cours, implémenté sous la forme d'un wrapper Jdb)
- DotNet (en cours, wrapper ICoreDebug)
- Windows CE/ARM/Ring3 via DebugAPI

Le wrapper Gdb permet de deboguer toute machine qui embarque un stub Gdb (les routeurs Cisco par exemple).

La complexité du développement réside dans les stubs (assembleur spécifique, programmation UART, ...) et dans modules de type OSHelpers qui doivent avoir une connaissance assez fine des structures et mécanismes internes du système d'exploitation supporté.

5 Conclusion

L'objectif de ce papier est de fournir un aperçu de l'architecture technique et des mécanismes mis en oeuvre dans un débogueur générique. GenDbg permet, via une interface commune, de déboguer tout type d'application sur n'importe quelle architecture. La conception modulaire de cet outil permet une distribution simplifiée ainsi que des évolutions rapides.

Il suffit par exemple de développer un nouveau stub et un module de désassemblage pour pouvoir l'utiliser dans sa version minimale sur une nouvelle architecture.

L'apport de nouvelles fonctionnalités telles que le multi-stubs ou encore l'emploi de scripts étend la puissance potentielle du framework au delà des logiciels de déboguage actuels. Cependant

GenDbg est loin d'être un produit fini et le chemin est long avant d'atteindre la maturité !

Références

1. http ://stenri.pisem.net
2. http ://www.microsoft.com/whdc/devtools/debugging/default.mspx
3. http ://www.gnu.org/software/gdb/
4. http ://www.ollydbg.de/
5. http ://www.sysersoft.com/
6. http ://www.immunitysec.com/products-immdbg.shtml
7. http ://rr0d.droids-corp.org/

A Script Logger.gds

```
dup CallArgs MainArgs
size MainArgs
if (%%LastResult%%!=2) goto Usage

; Créer le fichier de log
dup MainArgs.1 GlobalVars.LogFile
call Script\Lib.gds!InitLogFile GlobalVars.LogFile

; Initialisations
var GlobalVars.Indice int_le 0
```

```
var GlobalVars.Croix uchar[] "|/-\"
var HexData list
dup GlobalVars.LogFile HexData.File
var AppendData list
dup GlobalVars.LogFile AppendData.File

; BreakPoint sur les endroits ou on veut logger des choses
bpx 77D191A3
dup LastResult.Id BP_Log_1

:Loop
; Lancer le programme
g
dup LastResult BreakInfo
; on a un breakpoint -> faire tourner la croix
read uchar[] GlobalVars.Croix %%GlobalVars.Indice%% 1
print "#r%%LastResult%%"
var GlobalVars.Indice int_le (%%GlobalVars.Indice%%+1)%4
; Un BreakPoint est arrivé
if %%BreakInfo.BreakType%%!=2 goto DoAbort
if %%BreakInfo.BreakExtra%%==%%BP_Log_1%% goto Do_Log_1
goto Loop

;-------------------------------------------
:Do_Log_1
var LngBuff int_le @(@(esp+4)+40)
var BuffAddr int_le @(esp+4)+50
var State int_le @(@(esp+4)+1DCE4)
var NoSeq int_le @(@(esp+4)+1B394)
; Affichage
var AppendData.Txt uchar[] "INPUT_PROCESSING : State = %%State%%,
                                                  NoSeq = %%NoSeq%%,
                                                  sz = %%LngBuff%% #r#n"
call Script\Lib.gds!AppendInFile AppendData
; Affichage du buffer
read uchar[] %%BuffAddr%% %%LngBuff%%
dup LastResult HexData.Buffer
dup LngBuff HexData.Sz
var HexData.Tab uchar[] ""
call Script\Lib.gds!Hex2Ascii HexData
goto Loop

;-------------------------------------------
:DoAbort
bc *
print "Aborting ... (BreakType : %%BreakInfo.BreakType%%) #r#n"
```

```
return

;-------------------------------------------
:Usage
print "Usage %%MainArgs.0%% <LogFile>.#r#n"
return
```