Fuzz and Profit with WHVP

Damien Aumaitre daumaitre@quarkslab.com

Quarkslab

Résumé. Comment fuzzer du code kernel Windows avec la même facilité que lorsqu'on utilise libfuzzer [7] ? En 2017, Microsoft a introduit une API nommée WHVP (Windows Hypervisor Platform) permettant de contrôler finement et facilement des partitions Hyper-V. Cette présentation a pour objectif de partir à la découverte de cette API. Nous allons nous en servir pour créer des mini-vm spécialisées dans l'exercice d'une fonction particulière du noyau Windows (ou d'un de ses périphériques). Ces machines virtuelles vont nous permettre de créer plusieurs outils utiles à un chercheur de vulnérabilités comme par exemple des traceurs ou des fuzzers.

1 Introduction

Lors d'une étude d'un composant noyau (par exemple une fonctionnalité native du système d'exploitation), il est courant d'utiliser un déboggueur ainsi qu'un désassembleur.

On commence par faire le tour du propriétaire et l'expérience aidant on arrive très souvent sur une partie où le code mérite une analyse approfondie. Cette analyse peut se faire de manière manuelle mais risque d'être coûteuse en temps. On aimerait pouvoir attaquer facilement cette fonction, par exemple en la fuzzant.

Cependant le fuzzing de code kernel n'est pas une chose facile.

On peut par exemple installer le logiciel cible dans une machine virtuelle, puis écrire le fuzzer permettant d'exercer les fonctionnalités cibles. Pour monitorer la cible il faut configurer un déboggueur kernel (ou espérer qu'un crash dump suffira à l'analyse). Ensuite il faut lancer le fuzzer et être capable de restaurer un état sain de la cible une fois un crash détecté (par exemple en utilisant un snapshot).

Une fois un crash obtenu l'analyse peut ne pas être simple. En effet, la nature asynchrone d'un noyau rend la reproduction du crash difficile. Par exemple il peut y avoir une décorrélation entre le moment où on obtient

un crash et le testcase actuellement exécuté par le fuzzer (typique lors d'une corruption mémoire).

En comparaison l'état de l'art des fuzzers en userland est bien plus avancé. Grâce à des outils comme AFL [1], libfuzzer [7] ou honggfuzz [5], il suffit d'écrire et de compiler un binaire particulier appelé *harness* exercant la fonctionnalité choisie et de lancer le fuzzer.

Il est même possible d'attaquer des cibles sans avoir le code source en utilisant qemu ou QBDI [9].

Il existe des travaux utilisant syzcaller et kAFL [4] pour fuzzer le noyau Windows. L'approche retenue est globale et cible des points d'entrées externe du noyau (comme les appels système).

L'approche présentée dans cet article est différente. Elle est plus locale et cible une fonction précise. Elle montre comment en utilisant un hyperviseur (dans notre cas Hyper-V), il est possible de faire facilement du fuzzing d'API dans un kernel Windows.

2 WHVP (Windows Hypervisor Platform)

À la fin de l'année 2017, Microsoft a introduit une API pour piloter Hyper-V. Celle-ci se nomme WHVP (Windows Hypervisor Platform) [25]. Elle permet de créer des processeurs virtuels attachés à une partition (une machine virtuelle dans la terminologie MS) et de gérer les événements nécessitant une intervention de l'hyperviseur.

Cela permet d'avoir un contrôle très fin sur l'exécution de cette VM, il est possible de modifier le contexte d'exécution des processeurs virtuels (VP : *Virtual Processor*) ainsi que la mémoire physique des partitions (GPA : *Guest Physical Address*).

Il est à noter que l'API ne permet pas un contrôle total (comme le permettrait l'écriture de son propre hyperviseur) mais ceci est largement compensé par la facilité d'usage de celle-ci.

Microsoft a introduit cette API pour fournir à Oracle (VirtualBox) et VMWare la possibilité de faire fonctionner leurs solutions de virtualisation

au dessus d'Hyper-V. En effet Hyper-V devient incontournable dans la sécurité de Windows ce qui rend sa désactivation problématique.

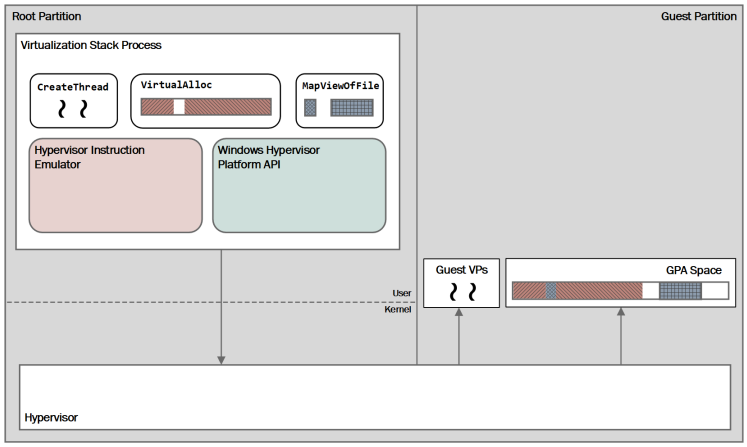

Fig. 1. Architecture WHVP (source Microsoft)

Plusieurs outils utilisant cette API ont déjà été publiés. On peut citer par exemple *applepie* [2] et *simpleator* [12]. L'hyperviseur *applepie* permet de remplacer le moteur d'émulation utilisé par *bochs* par Hyper-V tandis que *simpleator* a pour objectif d'exécuter la partie userland d'un binaire dans une partition Hyper-V afin de pouvoir étudier en toute sécurité du code malveillant.

L'approche retenue ici est différente. Elle consiste en l'exécution d'une fonction précise au sein d'une partition Hyper-V, permettant la création d'une vm minimaliste dédiée à cette fonction. Le contexte processeur et la mémoire de la partition peuvent être modifiés à volonté afin d'en faciliter l'analyse.

2.1 Découverte de l'API WHVP

Hyper-V est un hyperviseur de type 1 (bare metal). Le système d'exploitation principal est lui même dans une machine virtuelle appelée *Root Partition*.

L'API WHVP permet de communiquer avec l'hyperviseur pour créer d'autres partitions (machines virtuelles). L'utilisation de cette API est

assez simple.

Voici les étapes principales à effectuer :

— il faut d'abord vérifier si Hyper-V est présent avec la fonction `WHvGetCapability` ;
— il faut ensuite créer la partition avec la fonction `WHvCreatePartition` ;
— puis la configurer avec les fonctions `WHvSetupPartition` et `WHvSetPartitionProperty` ;
— il faut ensuite créer les processeurs virtuels (Virtual Processors ou Vp) avec la fonction `WHvCreateVirtualProcessor` ;
— la vm est enfin initialisée.

2.2 Cycle de vie d'un processeur virtuel

Le (ou les) processeurs virtuels sont démarrés à l'aide de la fonction `WHvRunVirtualProcessor`.

Cette fonction est bloquante et va exécuter la machine virtuelle jusqu'à l'apparition d'un événement qui va nécessiter une intervention de l'hyperviseur. Par exemple il peut s'agir de mémoire physique non présente dans la partition ou d'une faute du processeur virtuel.

Une fois l'événement traité, l'exécution du processeur virtuel est reprise avec la même fonction.

Il est possible de lire et écrire le contexte cpu du processeur virtuel avec les fonction `WHvGetVirtualProcessorRegisters` et `WHvSetVirtualProcessorRegisters`.

La mémoire physique de la machine virtuelle est allouée tout simplement avec les fonctions d'allocation de mémoire userland habituelles (`VirtualAlloc` par exemple) et partagée avec la machine virtuelle à l'aide de la fonction `WHvMapGpaRange`.

2.3 Exemple

Par défaut, le processeur virtuel démarre en mode réel 16 bits et va exécuter l'instruction située à l'adresse `0xffff0`.

Afin de montrer à quoi ressemble l'API, nous allons juste exécuter l'instruction `hlt` dans une partition Hyper-V, celle-ci va stopper immédiatement le processeur (cela va nous permettre de reprendre la main après l'exécution de l'instruction).

La première étape est d'allouer un buffer qui va contenir notre code.

```
UINT64 size = 0x1000;
LPVOID mem = VirtualAlloc(NULL, size, MEM_RESERVE | MEM_COMMIT,
    PAGE_READWRITE);
uint32_t addr = 0xfff0;
#define code "\xf4"  // [0xfff0] hlt
memcpy(&mem[addr], code, sizeof(code) - 1);
```

Ensuite nous créons une partition contenant un processeur virtuel.

```
HANDLE partition = INVALID_HANDLE_VALUE;
HRESULT hr = WHvCreatePartition(&partition);

WHV_PARTITION_PROPERTY partitionProperty;
partitionProperty.ProcessorCount = 1;

HRESULT hr = WHvSetPartitionProperty(partition,
                WHvPartitionPropertyCodeProcessorCount,
                &partitionProperty, sizeof(WHV_PARTITION_PROPERTY)
                    );

HRESULT hr = WHvSetupPartition(partition);
HRESULT hr = WHvCreateVirtualProcessor(partition, 0, 0);
```

Nous mappons au sein de la partition le buffer contenant notre code. Il faut noter l'existence ici de deux types d'adresses physiques (les *Host Physical Address* ou HPA et les *Guest Physical Address* ou GPA). Dans notre cas la HPA de notre buffer est `mem` et la GPA est `0xf0000`.

```
HRESULT hr = WHvMapGpaRange(partition, mem,
              0xf0000, 0x1000,
              WHvMapGpaRangeFlagRead | WHvMapGpaRangeFlagExecute);
```

Il nous reste juste à lancer l'exécution du processeur virtuel.

```
WHV_RUN_VP_EXIT_CONTEXT exitContext
HRESULT hr = WHvRunVirtualProcessor(partition, 0, &exitContext,
    sizeof(exitContext));
```

Celle-ci va retourner lorsque le processeur virtuel rencontre une condition nécessitant l'intervention de l'hyperviseur. Comme l'arrêt d'un processeur en est une, la fonction retourne. Nous pouvons ensuite vérifier que nous avons bien exécuté l'instruction `hlt` en vérifiant les champs de la structure `exitContext` ainsi que la valeur du registre `rip`.

```
WHV_REGISTER_NAME regs[] = {
    WHvX64RegisterCs,
    WHvX64RegisterRip,
    WHvX64RegisterRax,
};
WHV_REGISTER_VALUE values[sizeof(regs) / sizeof(regs[0])];
HRESULT hr = WHvGetVirtualProcessorRegisters(partition, 0, regs,
    3, values);
```

Dans l'exemple précédent nous avons utilisé du code C pour manipuler l'API.

Le premier prototype de ce projet a été réalisé en python. Comme nous le verrons plus tard, la vitesse d'exécution de la boucle d'événements de l'hyperviseur est très importante si l'on désire obtenir des performances suffisantes pour réaliser un fuzzer. Le coeur du projet a donc été réécrit en rust permettant d'améliorer grandement les performances, nous avons gardé une interface en python afin d'interagir facilement avec la majorité des outils de reverse déjà existants.

Le code a été séparé en 3 *crates* (modules en rust) :

— `whvp-sys` : bindings réalisés avec bindgen sur l'API native de WHVP ;
— `whvp-core` : coeur du projet contenant les interfaces permettant de s'intégrer avec un hyperviseur (pas forcément WHVP) ;
— `whvp-py` : bindings python réalisés avec pyO3 sur `whvp-core`.

Tous les exemples suivants utilisent ces bindings.

3 Exécution d'une fonction arbitraire

3.1 Exécution de code 64 bits

Même si exécuter du code 16 bits peut avoir un intérêt (par exemple on peut mapper et exécuter un BIOS), le cas d'usage majoritaire est d'avoir à étudier du code 64 bits.

Pour exécuter du code 64 bits dans notre partition nous avons deux moyens :

— démarrer en mode 16 bits puis exécuter un stub pour passer en mode 64 bits ;

— ou configurer le processeur virtuel pour qu'il démarre directement en mode 64 bits.

Nous avons privilégié la seconde solution qui est plus simple à mettre en oeuvre.

Il nous faut donc configurer le processeur virtuel pour qu'il démarre en mode 64 bits (appelé aussi *Long Mode* [11]).

Nous avons donc besoin de :

— configurer les registres de segments (en particulier activer le bit `Long` dans `cs`) ;
— configurer le registre `cr0` (pour activer la pagination) ;
— configurer le registre `cr4` (toujours pour la pagination) ;
— configurer le `MSR EFER` (pour activer le bit `LME` : *Long Mode Enable*) ;
— mettre en place une GDT et une IDT (avec bien évidemment des segments 64 bits) ;
— mettre en place des tables de pages et configurer le registre `cr3` pour pointer sur la table principale.

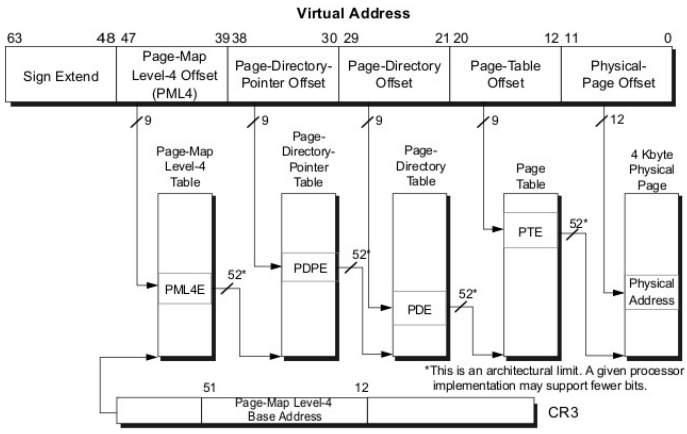

Fig. 2. Pagination

Cela fait beaucoup de configuration et nous n'avons pas encore exécuté le moindre octet.

Plutôt que de forger à la main tout le contexte nécessaire, il est plus facile de copier un contexte déjà configuré et donc valide.

On va donc se servir des informations fournies par le déboggueur noyau (par exemple Windbg), soit dynamiquement, soit à partir d'un dump mémoire ou tout autre moyen permettant d'avoir un contexte processeur noyau valide.

Après avoir configuré le processeur en mode 64 bits, nous sommes confrontés à un autre problème. Il faut copier le code à exécuter dans notre partition. Mais où copier ce code ?

En mode 64 bits la pagination est activée et le CPU manipule des adresses virtuelles (GVA ou Guest Virtual Address). De notre côté nous manipulons des GPA (Guest Physical Address). Nous avons donc besoin de pouvoir convertir les GVA en GPA. Cela passe par la configuration des tables de pages comme nous pouvons le voir sur la figure 2.

Heureusement pour nous, il existe un autre moyen pour mapper le code. En effet lorsqu'une page physique n'est pas mappée dans la partition, cela provoque une sortie de vm et l'exécution du processeur virtuel est interrompue.

L'idée est la suivante : nous configurons le processeur pour qu'il démarre en mode 64 bits. Dès que le processeur va avoir besoin d'accéder à une adresse virtuelle, il va parcourir les tables de pages qui sont nécessaires. Si celles-ci ne sont pas mappées, l'exécution s'interrompt et charge à nous de copier la page physique correspondante et de reprendre l'exécution. Ceci est vrai pour toutes les pages physiques, y compris les tables de pages.

Nous avons donc juste besoin d'un snapshot des pages physiques de notre cible avant son exécution. Pour les obtenir nous allons continuer d'utiliser notre déboggueur ou notre dump mémoire vu que toute l'information nécessaire y est disponible.

Nous avons tout ce qu'il faut pour démarrer l'exécution de notre fonction cible au sein d'une partition Hyper-V. Reste maintenant la

condition d'arrêt : comment savoir que notre fonction s'est exécutée ? Nous avons réalisé un fork du système d'exploitation, celui-ci va continuer son exécution dans notre environnement mais ce n'est pas ce qui nous intéresse.

Une possibilité est d'insérer un point d'arrêt logiciel sur l'adresse de retour de notre fonction lors du mapping des pages physiques lues du snapshot. Lorsque le processeur virtuel va exécuter l'adresse de retour, l'hyperviseur va prendre la main et nous pourrons interrompre l'exécution de notre fonction.

Pour résumer nous avons besoin de 2 choses : un contexte processeur et un ensemble de pages physiques. Ces 2 éléments sont facilement obtenables à partir d'un déboggueur noyau.

3.2 Cas pratique

Prenons comme exemple l'exécution de la fonction `RtlInitUnicodeString`.

Son code désassemblé est le suivant :

```
kd> uf @rip
nt!RtlInitUnicodeString:
fffff805`78c45b50 48c70100000000  mov     qword ptr [rcx],0
fffff805`78c45b57 48895108        mov     qword ptr [rcx+8],rdx
fffff805`78c45b5b 4885d2          test    rdx,rdx
fffff805`78c45b5e 7501            jne     nt!RtlInitUnicodeString+0x11 (fffff805`78c45b61)
       Branch

nt!RtlInitUnicodeString+0x10:
fffff805`78c45b60 c3              ret  Branch

nt!RtlInitUnicodeString+0x11:
fffff805`78c45b61 48c7c0ffffffff  mov     rax,0FFFFFFFFFFFFFFFFh
fffff805`78c45b68 0f1f840000000000 nop     dword ptr [rax+rax]

nt!RtlInitUnicodeString+0x20:
fffff805`78c45b70 48ffc0          inc     rax
fffff805`78c45b73 66833c4200      cmp     word ptr [rdx+rax*2],0
fffff805`78c45b78 75f6            jne     nt!RtlInitUnicodeString+0x20 (fffff805`78c45b70)
       Branch

nt!RtlInitUnicodeString+0x2a:
fffff805`78c45b7a 4803c0          add     rax,rax
fffff805`78c45b7d 483dfeff0000    cmp     rax,0FFFEh
fffff805`78c45b83 0f839d911b00    jae     nt!RtlInitUnicodeString+0x1b91d6 (fffff805`78
     dfed26)  Branch

nt!RtlInitUnicodeString+0x39:
fffff805`78c45b89 668901          mov     word ptr [rcx],ax
fffff805`78c45b8c 6683c002        add     ax,2
fffff805`78c45b90 66894102        mov     word ptr [rcx+2],ax
fffff805`78c45b94 c3              ret

nt!RtlInitUnicodeString+0x1b91d6:
fffff805`78dfed26 b8fcff0000      mov     eax,0FFFCh
fffff805`78dfed2b e9596ee4ff      jmp     nt!RtlInitUnicodeString+0x39 (fffff805`78c45b89)
       Branch
```

Comment pourrait-on faire pour l'exécuter dans notre partition ?

La première chose à effectuer est de récupérer le contexte initial auprès d'un déboggueur noyau. Nous avons choisi d'exposer une interface sur celui-ci en utilisant *pykd* et *rpyc*. *pykd* [8] est une extension pour Windbg permettant d'interagir en python avec celui-ci. *rpyc* [10] est un framework pour réaliser facilement des RPC (Remote Procedure Call) en python. En combinant les deux nous pouvons à distance contrôler Windbg. Nous nous en servons pour récupérer le contexte du processeur ainsi que les pages physiques à mapper dans la partition.

```
import time
import whvp

from whvp.snapshot import RpycSnapshot

whvp.init_log()

hostname = "localhost"
port = 18861

snapshot = RpycSnapshot(hostname, port)
emulator = whvp.Emulator()

context = snapshot.get_initial_context()
```

Il faut ensuite configurer les registres du processeur virtuel. Tout se passe comme si le système d'exploitation était *forké* dans la partition.

```
# GDT
emulator.set_table_reg("gdt", context["gdtr"], context["gdtl"])

# IDT
emulator.set_table_reg("idt", context["idtr"], context["idtl"])

# CR0
emulator.set_reg("cr0", context["cr0"])

# CR3
emulator.set_reg("cr3", context["cr3"])

# CR4
emulator.set_reg("cr4", context["cr4"])

# IA32_EFER
emulator.set_reg("efer", context["efer"])

emulator.set_segment_reg("cs", 0, 0, 1, 0, context["cs"])
emulator.set_segment_reg("ss", 0, 0, 0, 0, context["ss"])
emulator.set_segment_reg("ds", 0, 0, 0, 0, context["ds"])
emulator.set_segment_reg("es", 0, 0, 0, 0, context["es"])
```

```
emulator.set_segment_reg("fs", context["fs_base"], 0, 0, 0,
    context["fs"])
emulator.set_segment_reg("gs", context["gs_base"], 0, 0, 0,
    context["gs"])

emulator.set_reg("rax", context["rax"])
emulator.set_reg("rbx", context["rbx"])
emulator.set_reg("rcx", context["rcx"])
emulator.set_reg("rdx", context["rdx"])
emulator.set_reg("rsi", context["rsi"])
emulator.set_reg("rdi", context["rdi"])
emulator.set_reg("r8", context["r8"])
emulator.set_reg("r9", context["r9"])
emulator.set_reg("r10", context["r10"])
emulator.set_reg("r11", context["r11"])
emulator.set_reg("r12", context["r12"])
emulator.set_reg("r13", context["r13"])
emulator.set_reg("r14", context["r14"])
emulator.set_reg("r15", context["r15"])

emulator.set_reg("rbp", context["rbp"])
emulator.set_reg("rsp", context["rsp"])

emulator.set_reg("rip", context["rip"])

emulator.set_reg("rflags", context["rflags"])

return_address = context["return_address"]
```

Lorsque le processeur virtuel a besoin d'une page physique qui n'est pas présente dans la mémoire de la partition, il faut être capable de lui fournir les données contenues dans celle-ci. D'une façon analogue au contexte, nous utilisons aussi le déboggueur noyau pour cela.

```
def memory_access_callback(gpa, gva):
    data = snapshot.memory_access_callback(gpa)
    if data:
        return data
    else:
        raise Exception(F"no data for gpa {gpa:X} (gva {gva:x})")
```

Nous avons maintenant toutes les informations nécessaires pour exécuter la fonction dans la partition.

```
params = {
    "coverage_mode": "no",
    "return_address": return_address,
    "save_context": False,
    "display_instructions": False,
    "save_instructions": False,
    "display_vm_exits": True,
    "stopping_addresses": [],
}
```

```
whvp.log("running emulator")
start = time.time()
result = emulator.run_until(params, memory_access_callback=
    memory_access_callback)
end = time.time()
whvp.log(F"{result} in {end - start:.2f} secs")
```

Nous arrivons à exécuter correctement la fonction :

```
> python .\samples\RtlInitUnicodeString.py
2020-01-30 11:53:31,284 INFO  [whvp] running emulator
2020-01-30 11:53:31,349 INFO  [whvp::core] used 8.19 kB for code and 36.86 kB for data
2020-01-30 11:53:31,349 INFO  [whvp::core] got 12 vm exits (Success)
2020-01-30 11:53:31,349 INFO  [whvp] vm exit: 12, coverage 1, status Success in 0.06
    secs
2020-01-30 11:53:31,354 DEBUG [whvp::whvp] destructing partition
2020-01-30 11:53:31,357 DEBUG [whvp::mem] destructing allocator
```

Nous avons obtenu 12 sorties de vm.

La première sortie de vm correspond au fait que la page physique `0x58147f80` n'est pas présente.

```
MemoryAccess(
    VpContext {
        Rip: fffff80578c45b50,
        Rflags: 0000000000050246,
    },
    MemoryAccessContext {
        AccessInfo: MemoryAccessInfo {
            AccessType: Write,
            GpaUnmapped: true,
            GvaValid: false,
        },
        Gpa: 0000000058147f80,
        Gva: 0000000000000000,
    },
)
```

Nous voyons aussi que l'adresse qui est en train d'être exécutée est `0xfffff80578c45b50`. En convertissant cette adresse vers son adresse physique, nous comprenons rapidement pourquoi cette adresse physique est nécessaire, il s'agit de la table de page principale pointée par le registre `cr3`.

```
kd> r cr3
cr3=0000000058147002
```

Les 4 sorties de vm suivantes correspondent aux différentes tables de pages nécessaires à la conversion de l'adresse virtuelle de `rip`.

```
MemoryAccess(
    VpContext {
        Rip: fffff80578c45b50,
        Rflags: 0000000000050246,
    },
    MemoryAccessContext {
        AccessInfo: MemoryAccessInfo {
            AccessType: Write,
            GpaUnmapped: true,
            GvaValid: false,
        },
        Gpa: 00000000011080a8,
        Gva: 0000000000000000,
    },
)
MemoryAccess(
    VpContext {
        Rip: fffff80578c45b50,
        Rflags: 0000000000050246,
    },
    MemoryAccessContext {
        AccessInfo: MemoryAccessInfo {
            AccessType: Write,
            GpaUnmapped: true,
            GvaValid: false,
        },
        Gpa: 0000000001109e30,
        Gva: 0000000000000000,
    },
)
MemoryAccess(
    VpContext {
        Rip: fffff80578c45b50,
        Rflags: 0000000000050246,
    },
    MemoryAccessContext {
        AccessInfo: MemoryAccessInfo {
            AccessType: Write,
            GpaUnmapped: true,
            GvaValid: false,
        },
        Gpa: 0000000001113228,
        Gva: 0000000000000000,
    },
)
MemoryAccess(
    VpContext {
        Rip: fffff80578c45b50,
        Rflags: 0000000000050246,
    },
    MemoryAccessContext {
        AccessInfo: MemoryAccessInfo {
            AccessType: Execute,
            GpaUnmapped: true,
            GvaValid: true,
        },
        Gpa: 0000000001f3db50,
```

```
            Gva: fffff80578c45b50,
        },
    )
```

Comme nous pouvons le voir en interrogeant le déboggueur :

```
kd> !pte @rip
                                            VA fffff80578c45b50
PXE at FFFFFC7E3F1F8F80    PPE at FFFFFC7E3F1F00A8    PDE at FFFFFC7E3E015E30    PTE at
     FFFFFC7C02BC6228
contains 0000000001108063  contains 0000000001109063  contains 0000000001113063  contains
     0900000001F3D021
pfn 1108      ---DA--KWEV  pfn 1109      ---DA--KWEV  pfn 1113      ---DA--KWEV  pfn 1f3d
          ----A--KREV
```

La première instruction de la fonction est une écriture à l'adresse pointée par `rcx` (la structure `_UNICODE_STRING`).

```
fffff805`78c45b50 48c70100000000  mov     qword ptr [rcx],0
```

D'une manière similaire aux sorties de vm précédentes nous avons aussi 4 sorties de vm correspondant à la traduction de l'adresse virtuelle contenue dans `rcx`.

```
    MemoryAccess(
        VpContext {
            Rip: fffff80578c45b50,
            Rflags: 0000000000050246,
        },
        MemoryAccessContext {
            AccessInfo: MemoryAccessInfo {
                AccessType: Write,
                GpaUnmapped: true,
                GvaValid: false,
            },
            Gpa: 00000000005a6190,
            Gva: 0000000000000000,
        },
    )
    MemoryAccess(
        VpContext {
            Rip: fffff80578c45b50,
            Rflags: 0000000000050246,
        },
        MemoryAccessContext {
            AccessInfo: MemoryAccessInfo {
                AccessType: Write,
                GpaUnmapped: true,
                GvaValid: false,
            },
            Gpa: 00000000005a7928,
            Gva: 0000000000000000,
        },
    )
    MemoryAccess(
```

```
      VpContext {
          Rip: fffff80578c45b50,
          Rflags: 0000000000050246,
      },
      MemoryAccessContext {
          AccessInfo: MemoryAccessInfo {
              AccessType: Write,
              GpaUnmapped: true,
              GvaValid: false,
          },
          Gpa: 0000000068387ff0,
          Gva: 0000000000000000,
      },
  )
  MemoryAccess(
      VpContext {
          Rip: fffff80578c45b50,
          Rflags: 0000000000050246,
      },
      MemoryAccessContext {
          AccessInfo: MemoryAccessInfo {
              AccessType: Write,
              GpaUnmapped: true,
              GvaValid: true,
          },
          Gpa: 0000000013474f90,
          Gva: ffffef8ca4bfef90,
      },
  )
```

```
kd> r @rcx
rcx=ffffef8ca4bfef90
kd> !pte @rcx
                                           VA ffffef8ca4bfef90
PXE at FFFFFC7E3F1F8EF8    PPE at FFFFFC7E3F1DF190    PDE at FFFFFC7E3BE32928    PTE at
     FFFFFC77C6525FF0
contains 0A000000005A6863  contains 0A000000005A7863  contains 0A00000068387863  contains
     8A00000013474863
pfn 5a6       ---DA--KWEV  pfn 5a7       ---DA--KWEV  pfn 68387     ---DA--KWEV  pfn 13474
          ---DA--KW-V
```

Les sorties de vm restantes correspondent à l'exécution de l'adresse placée sur la pile lors de l'appel à `RtlInitUnicodeString`, à savoir l'adresse `0fffff805792000dd`.

```
  MemoryAccess(
      VpContext {
          Rip: fffff805792000dd,
          Rflags: 0000000000050246,
      },
      MemoryAccessContext {
          AccessInfo: MemoryAccessInfo {
              AccessType: Write,
              GpaUnmapped: true,
              GvaValid: false,
          },
          Gpa: 0000000001116000,
```

```
            Gva: 0000000000000000,
        },
    )
    MemoryAccess(
        VpContext {
            Rip: fffff805792000dd,
            Rflags: 0000000000050246,
        },
        MemoryAccessContext {
            AccessInfo: MemoryAccessInfo {
                AccessType: Execute,
                GpaUnmapped: true,
                GvaValid: true,
            },
            Gpa: 00000000213f80dd,
            Gva: fffff805792000dd,
        },
    )
    Exception(
        VpContext {
            Rip: fffff805792000dd,
            Rflags: 0000000000040246,
        },
        ExceptionContext {
            ExceptionType: 3,
        },
    )

kd> k
# Child-SP          RetAddr           Call Site
00 ffffef8c`a4bfef68 fffff805`792000dd nt!RtlInitUnicodeString
```

Pour savoir si nous avons exécuté l'adresse de retour nous avons juste mis un point d'arrêt à cette adresse (d'où l'interruption 3).

Au final nous obtenons une vm minimaliste qui exerce uniquement le code qui nous intéresse.

Une fois les pages mappées l'exécution de la machine virtuelle est plus rapide, car nous n'avons plus besoin de mapper les pages physiques manquantes dans la partition. Son contexte processeur et sa mémoire sont manipulables à volonté. De plus on part d'un état connu et déterministe.

3.3 Contexte processeur

Dans l'état actuel on connaît le contexte de départ de notre fonction et le contexte lorsque la fonction retourne (si évidemment elle retourne).

Il serait plus utile d'avoir le contexte sur chaque instruction.

Comme on contrôle le contexte de départ, un moyen simple pour obtenir le contexte serait d'activer le flag *TF (Trap Flag)* dans le registre `rflags`. Celui-ci va indiquer au processeur qu'il doit générer une interruption avant l'exécution de chaque instruction. Il faut donc aussi configurer la partition pour obtenir des sorties de vm lorsque l'interruption 1 est déclenchée dans la vm.

Nous avons aussi la possibilité de configurer les registres de debug (`Dr`) pour placer des points d'arrêt dans le code exécuté dans la partition.

Sur chaque interruption il est possible de récupérer le contexte du processeur virtuel. On obtient donc une trace d'exécution de notre fonction.

Le gros avantage par rapport à l'usage d'un déboggueur noyau classique est que la trace est déterministe. Chaque exécution avec les mêmes paramètres de départ (registres et mémoire) va produire la même trace.

De plus il est aisé de comparer 2 traces d'exécution, l'ASLR n'intervenant pas par exemple.

3.4 Restauration de la mémoire

Si on veut une exécution déterministe il faut aussi pouvoir être capable de restaurer les pages de la vm.

Une manière inefficace serait de libérer toutes les pages physiques de la partition et de les remapper comme pour la première exécution.

L'inconvénient est qu'on perd tout l'avantage de notre vm spécialisée, à chaque exécution il faudra récupérer les pages physiques auprès du déboggueur et les mapper au sein de la partition.

Heureusement pour nous, l'API WHVP permet de monitorer les pages qui ont été modifiées durant un laps de temps. Il suffit donc à la fin de l'exécution de notre fonction d'interroger l'hyperviseur avec la fonction `WHvQueryGpaRangeDirtyBitmap` et de restaurer les pages physiques qui ont été modifiées.

Par exemple prenons la fonction `ExAllocatePoolWithTag`. Celle-ci va changer l'état interne du noyau pour refléter l'allocation. À chaque

exécution de la fonction nous obtenons une valeur différente pour le registre `rax`. Celui-ci contient l'adresse du buffer alloué.

```
> python .\samples\ExAllocatePoolWithTag.py
2020-01-30 14:43:47,540 INFO  [whvp] running emulator
2020-01-30 14:43:47,630 INFO  [whvp::core] used 28.67 kB for code and 139.26 kB for data
2020-01-30 14:43:47,631 INFO  [whvp::core] got 42 vm exits (Success)
2020-01-30 14:43:47,631 INFO  [whvp] vm exit: 42, coverage 1, status Success in 0.09 secs
2020-01-30 14:43:47,632 INFO  [whvp] rax is ffffcb0ea0b03410
2020-01-30 14:43:47,632 INFO  [whvp] running emulator
2020-01-30 14:43:47,637 INFO  [whvp::core] used 28.67 kB for code and 143.36 kB for data
2020-01-30 14:43:47,637 INFO  [whvp::core] got 2 vm exits (Success)
2020-01-30 14:43:47,637 INFO  [whvp] vm exit: 2, coverage 1, status Success in 0.00 secs
2020-01-30 14:43:47,638 INFO  [whvp] rax is ffffcb0ea0b05610
2020-01-30 14:43:47,642 DEBUG [whvp::whvp] destructing partition
2020-01-30 14:43:47,644 DEBUG [whvp::mem] destructing allocator
```

Remarquons que le nombre de sorties de vm est bien inférieur lors de la seconde exécution. Faisons le même test en restaurant la mémoire entre chaque appel.

```
for i in range(5):
    whvp.log("running emulator")
    set_context(context)
    start = time.time()
    result = emulator.run_until(params, memory_access_callback=
        memory_access_callback)
    end = time.time()
    whvp.log(F"{result} in {end - start:.2f} secs")

    rax = emulator.get_reg("rax")
    whvp.log(F"rax is {rax:x}")

    emulator.restore_snapshot()
```

Cette fois, nous obtenons à chaque exécution la même adresse pour le buffer.

```
  > python .\samples\ExAllocatePoolWithTag.py
2020-01-30 14:48:39,965 INFO  [whvp] running emulator
2020-01-30 14:48:40,047 INFO  [whvp::core] used 28.67 kB for code
    and 139.26 kB for data
2020-01-30 14:48:40,047 INFO  [whvp::core] got 42 vm exits (Success)
2020-01-30 14:48:40,048 INFO  [whvp] vm exit: 42, coverage 1, status
     Success in 0.08 secs
2020-01-30 14:48:40,048 INFO  [whvp] rax is ffffcb0ea0b03410
2020-01-30 14:48:40,049 INFO  [whvp] running emulator
2020-01-30 14:48:40,050 INFO  [whvp::core] used 28.67 kB for code
    and 139.26 kB for data
2020-01-30 14:48:40,051 INFO  [whvp::core] got 1 vm exits (Success)
2020-01-30 14:48:40,051 INFO  [whvp] vm exit: 1, coverage 1, status
    Success in 0.00 secs
2020-01-30 14:48:40,051 INFO  [whvp] rax is ffffcb0ea0b03410
2020-01-30 14:48:40,052 INFO  [whvp] running emulator
2020-01-30 14:48:40,053 INFO  [whvp::core] used 28.67 kB for code
    and 139.26 kB for data
2020-01-30 14:48:40,054 INFO  [whvp::core] got 1 vm exits (Success)
```

```
2020-01-30 14:48:40,054 INFO  [whvp] vm exit: 1, coverage 1, status
    Success in 0.00 secs
2020-01-30 14:48:40,055 INFO  [whvp] rax is ffffcb0ea0b03410
2020-01-30 14:48:40,056 INFO  [whvp] running emulator
2020-01-30 14:48:40,058 INFO  [whvp::core] used 28.67 kB for code
    and 139.26 kB for data
2020-01-30 14:48:40,058 INFO  [whvp::core] got 1 vm exits (Success)
2020-01-30 14:48:40,059 INFO  [whvp] vm exit: 1, coverage 1, status
    Success in 0.00 secs
2020-01-30 14:48:40,060 INFO  [whvp] rax is ffffcb0ea0b03410
2020-01-30 14:48:40,060 INFO  [whvp] running emulator
2020-01-30 14:48:40,062 INFO  [whvp::core] used 28.67 kB for code
    and 139.26 kB for data
2020-01-30 14:48:40,062 INFO  [whvp::core] got 1 vm exits (Success)
2020-01-30 14:48:40,063 INFO  [whvp] vm exit: 1, coverage 1, status
    Success in 0.00 secs
2020-01-30 14:48:40,063 INFO  [whvp] rax is ffffcb0ea0b03410
2020-01-30 14:48:40,067 DEBUG [whvp::whvp] destructing partition
2020-01-30 14:48:40,069 DEBUG [whvp::mem] destructing allocator
```

Pouvoir fixer l'état du noyau avant chaque exécution est très pratique, notamment pour étudier une vulnérabilité par exemple. Il est assez trivial dans ce cas de monitorer les allocations pour déterminer le lieu de la corruption mémoire.

4 Fuzzer

Actuellement nous sommes capables d'exécuter autant de fois que nous voulons une fonction noyau arbitraire en maîtrisant à la fois le contexte CPU de départ ainsi que la mémoire de la vm.

Nous sommes aussi capable d'avoir une couverture du code exécuté. L'idée est donc de développer un fuzzer guidé à partir d'un snapshot mémoire (*snapshot based coverage guided fuzzer*).

Le principe d'un fuzzer est d'exécuter quelque chose avec des entrées qui n'ont pas été prévues par le développeur pour ensuite détecter les comportements anormaux.

Pour déterminer le format des paramètres de la fonction il s'agit de faire une étude préalable (de toute façon nécessaire pour de l'audit de code) sur les entrées contrôlées par l'attaquant.

Un comportement anormal va être dans notre cas l'exécution d'une fonction qui n'est pas prévue. Comme on exécute du code noyau un candidat naturel va être `KeBugCheck` mais rien ne nous oblige à se

restreindre.

Nous avons 2 listes de fonctions : la première va contenir les addresses de retour **normales**, la seconde va contenir les adresses des fonctions **anormales** indiquant ainsi que la fonction ne s'est pas exécutée comme prévue. Nous allons considérer ceci comme un *crash* et enregistrer dans un fichier les paramètres ayant abouti à cette exécution anormale.

Lors du mapping des pages physiques, nous insérons des points d'arrêt logiciel sur les adresses contenues dans ces 2 listes de fonctions. La résolution des adresses est facilitée car nous sommes déjà connecté à un déboggueur, il nous suffit juste de l'interroger pour obtenir les adresses qui nous intéressent.

Un fuzzer assisté par de la couverture de code fonctionne de la manière suivante :

— la cible est instrumentée pour récupérer la couverture de code ;
— un corpus d'entrées est chargé ;
— une entrée du corpus est choisie ;
— puis mutée ;
— la cible est exécutée ;
— si une nouvelle couverture de code est obtenue, la nouvelle entrée est rajoutée au corpus.

Nous sommes donc plus intéressés par la découverte de nouveaux chemins plus que par le contexte processeur complet à chaque exécution d'une instruction.

Nous allons donc obtenir notre couverture de code autrement. Au lieu de mapper directement les pages de code, nous allons mapper des pages remplies d'octets `0xcc` (qui correspond à l'instruction `int 3`). A chaque interruption, nous notons l'adresse et nous restaurons l'instruction originale avant de reprendre l'exécution. Nous aurons donc la couverture de code sur les instructions uniques exécutées par le processeur. Ce compromis est acceptable vis-à-vis du gain en performance.

Nous obtiendrons ainsi une sortie de vm uniquement sur l'exécution de nouvelles branches de code et les performances vont grandement s'améliorer.

La trace sera moins précise mais beaucoup plus rapide. Et la rapidité est très importante pour un fuzzer.

Pour le choix de l'entrée, comme il s'agit de l'analyse précise d'une fonction particulière, il est légitime de penser que nous connaissons le prototype de la fonction ainsi que les paramètres contrôlés par l'attaquant. Nous allons nous contenter de voir cette entrée comme un buffer avec une adresse et une taille. Les données originales sont lues puis mutées avant d'être réécrites en mémoire. Pour la mutation nous utilisons des heuristiques classiques de mutation comme celles utilisés dans *radamsa* ou *AFL*.

La stratégie pour sélectionner une entrée du corpus est minimaliste. Nous nous contentons de tourner parmi les entrées provoquant une nouvelle couverture de code.

Il est aussi utile de monitorer le répertoire contenant le corpus, dès qu'un nouveau fichier est rajouté dans ce répertoire il est automatiquement ajouté à la liste de travail du fuzzer et sélectionné comme point de départ d'une future mutation.

Pour reproduire un crash (par exemple pour obtenir une trace avec la couverture de code complète), il suffit de réécrire l'entrée avec l'entrée ayant provoqué le crash et ensuite d'exécuter la cible.

Par contre cela se complique pour reproduire le crash en dehors de l'outil. En effet comme lorsqu'on utilise *libfuzzer*, on attaque une fonction bas-niveau (au niveau de l'API). La reproduction nécessite donc d'injecter l'entrée qui provoque le crash dans un client dédié ou tout autre moyen permettant d'appeler la fonction cible. Nous considérons que ce client est un préalable à l'analyse (de toute façon nécessaire pour être capable d'utiliser le déboggueur noyau).

Le triage des crashs est facilité par la stabilité des adresses. Chaque crash est rejoué (en dehors du fuzzer) et ceux finissant par les mêmes adresses sont considérés comme des duplicata. Le choix de la taille à considérer est bien évidemment à ajuster pour chaque cible.

4.1 Cas pratique

Dans le cadre d'une formation interne nous avons développé un driver noyau reproduisant une vulnérabilité découverte [26] au sein du parseur utilisé par le composant KSE (Kernel Shim Engine) [15] du noyau Windows.

Ce driver est idéal pour tester notre fuzzer. En effet nous sommes sûrs d'avoir une vulnérabilité à trouver, cette vulnérabilité se trouve dans du code qui effectue du parsing (donc particulièrement pénible à auditer).

Concrètement le driver fonctionne de la manière suivante :

— il charge et mappe en mémoire le fichier décrivant les *shims* ;
— il fournit un *ioctl* permettant de chercher un *shim* par son nom ;
— lors de la recherche du *shim*, il parse le fichier mappé en mémoire.

Notre stratégie va être la suivante :

— grâce à un déboggueur noyau nous posons un point d'arrêt sur le handler de cet IOCTL. Celui-ci nous permettra d'avoir accès au contexte du processeur et à la mémoire de notre cible ;
— nous créeons alors un processeur virtuel avec l'api WHVP en lui fournissant comme contexte de départ le contexte CPU du déboggueur ;
— le processeur virtuel est alors démarré ;
— à l'issue de cette première exécution, tout le code nécessaire à l'exécution de notre fonction cible est présent en mémoire.

Nous pouvons maintenant attaquer la phase de fuzzing :

— le contexte CPU et mémoire est restauré ;
— nous mutons le buffer contenant les données des *shims* ;
— nous lançons l'exécution du processeur virtuel ;
— si la couverture obtenue par celle-ci est différente de la couverture de code initial nous stockons les données mutées ainsi que le coverage obtenu ;
— nous faisons ça pour chaque membre de notre corpus ;
— les buffers ayant obtenu le meilleur coverage sont utilisés comme point de départ des mutations ;

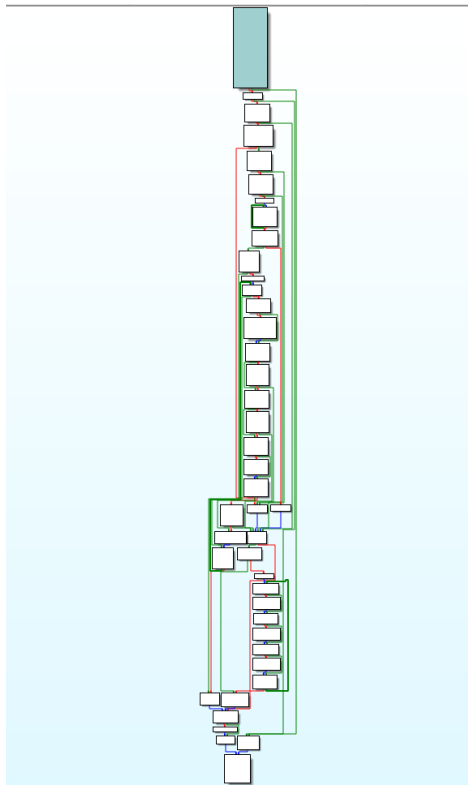

Fig. 3. Point d'entrée du parseur

— on recommence jusqu'à exécuter soit la fonction `KeBugCheck` soit l'adresse de retour.

4.2 Résultats

La première étape est de vérifier que le code exercé par la fonction cible s'exécute correctement dans notre partition.

```
> whvp-tracer --coverage instrs --workdir .\tmp\traces\sstic
2020-01-30 17:47:40,323 INFO  [whvp] resolving forbidden addresses
2020-01-30 17:47:40,337 INFO  [whvp] getting initial context
2020-01-30 17:47:40,373 INFO  [whvp] tracing SdbParser!DriverDispatchIoctl_GetNbShim+0x0 (
     fffff8057d1b653c)
2020-01-30 17:47:40,375 INFO  [whvp] running emulator
2020-01-30 17:47:42,154 INFO  [whvp] vm exit: 59206, coverage 59121, status Success in
     1.78 secs
2020-01-30 17:47:42,160 DEBUG [whvp::whvp] destructing partition
2020-01-30 17:47:42,163 DEBUG [whvp::mem] destructing allocator
```

La fonction est relativement petite (environ 60000 instructions). Il est à remarquer que l'exécution est lente (environ 1.8s). En effet chaque

vm-exit est coûteuse, il vaut mieux en faire le moins possible. Nous n'avons pas besoin d'avoir une couverture de code parfaite, nous sommes plus intéressés par la découverte de nouveaux chemins.

Le buffer non modifié est le suivant :

```
00000000  03 00 00 00 00 00 00 00  73 64 62 66 02 78 a2 00  |........sdbf.x..|
00000010  00 00 03 78 14 00 00 00  02 38 07 70 03 38 01 60  |...x.....8.p.8.`|
00000020  16 40 01 00 00 00 01 98  00 00 00 00 03 78 0e 00  |.@...........x..|
00000030  00 00 02 38 17 70 03 38  01 60 01 98 00 00 00 00  |...8.p.8.`......|
00000040  03 78 0e 00 00 00 02 38  07 70 03 38 04 90 01 98  |.x.....8.p.8....|
00000050  00 00 00 00 03 78 20 00  00 00 02 38 1c 70 03 38  |.....x ....8.p.8|
00000060  01 60 16 40 02 00 00 00  01 98 0c 00 00 00 53 59  |.`.@..........SY|
00000070  53 2e 54 52 50 32 e8 00  00 00 03 78 14 00 00 00  |S.TRP2.....x....|
00000080  02 38 1c 70 03 38 0b 60  16 40 02 00 00 00 01 98  |.8.p.8.`.@......|
00000090  00 00 00 00 03 78 1a 00  00 00 02 38 25 70 03 38  |.....x.....8%p.8|
000000a0  01 60 01 98 0c 00 00 00  45 4c 54 54 49 4c 59 4d  |.`......ELTTILYM|
000000b0  ba 00 00 00 01 70 a0 00  00 00 25 70 28 00 00 00  |.....p....%p(...|
000000c0  01 60 06 00 00 00 10 90  10 00 00 00 83 ed ea c7  |.`..............|
000000d0  d7 34 1c 45 b2 1f df 72  6d c7 c2 07 17 40 02 00  |.4.E...rm....@..|
000000e0  00 00 03 60 26 00 00 00  1c 70 6c 00 00 00 01 60  |...`&....pl....`|
000000f0  48 00 00 00 01 60 26 00  00 00 01 60 68 00 00 00  |H....`&....`h...|
00000100  04 90 10 00 00 00 78 56  34 12 34 12 78 56 12 34  |......xV4.4.xV.4|
00000110  56 78 12 34 56 78 08 70  06 00 00 00 01 60 06 01  |Vx.4Vx.p.....`..|
00000120  00 00 06 50 ff ff ff ff  02 00 01 00 26 70 28 00  |...P........&p(.|
00000130  00 00 01 60 06 00 00 00  10 90 10 00 00 00 83 ed  |...`............|
00000140  ea c7 d7 34 1c 45 b2 1f  df 72 6d c7 c2 07 17 40  |...4.E...rm....@|
00000150  02 00 00 00 03 60 26 00  00 00 01 78 70 00 00 00  |.....`&....xp...|
00000160  01 88 1a 00 00 00 6d 00  79 00 4c 00 69 00 74 00  |......m.y.L.i.t.|
00000170  74 00 6c 00 65 00 53 00  68 00 69 00 6d 00 00 00  |t.l.e.S.h.i.m...|
00000180  01 88 1c 00 00 00 53 00  68 00 69 00 6d 00 4b 00  |......S.h.i.m.K.|
00000190  65 00 79 00 6c 00 6f 00  67 00 67 00 65 00 72 00  |e.y.l.o.g.g.e.r.|
000001a0  00 00 01 88 1a 00 00 00  69 00 38 00 30 00 34 00  |........i.8.0.4.|
000001b0  32 00 70 00 72 00 74 00  2e 00 73 00 79 00 73 00  |2.p.r.t...s.y.s.|
000001c0  00 00 01 88 08 00 00 00  61 00 61 00 61 00 00 00  |........a.a.a...|
```

Toujours pour des raisons de performances nous allons éviter de modifier l'intégralité du buffer et nous contenter de muter uniquement les `0x60` premiers octets.

Après avoir lancé le fuzzer, nous obtenons très rapidement un crash. Le nombre d'exécution par seconde est aussi très satisfaisant (les exemples sont réalisés sur un laptop Lenovo T470 avec juste un coeur utilisé). Nous voyons que nous obtenons environ 2000 exec/s et qu'un premier crash tombe en quelques secondes après un peu plus de 8000 itérations. Le compromis pour obtenir la couverture de code uniquement sur les nouvelles instructions est payant. Nous passons d'une exécution toutes les 2 secondes à 2000 exécutions par secondes.

```
> whvp-fuzzer --coverage hit --input "poi(poi(poi(sdbparser!SdbHandle)+8)+8)" --input-size
      0x60 --workdir .\tmp\traces\sstic --stop-on-crash
2020-01-30 16:39:35,229 INFO  [whvp] fuzzing SdbParser!DriverDispatchIoctl_GetNbShim+0x0 (
     fffff8057d1b653c)
2020-01-30 16:39:35,229 INFO  [whvp] setting forbidden addresses
2020-01-30 16:39:35,246 INFO  [whvp] input is ffff898319d03d20
2020-01-30 16:39:35,247 INFO  [whvp] input size is 60
2020-01-30 16:39:35,248 INFO  [whvp] fuzzer workdir is .\tmp\traces\sstic\fuzz\947f7fd2
     -3183-4b9b-87d3-df3b7815e210
2020-01-30 16:39:35,250 INFO  [whvp::core] first executing a full trace to map memory
2020-01-30 16:39:36,877 INFO  [whvp::core] vm exit: 2992, coverage 2913, status Success
```

```
2020-01-30 16:39:36,878 INFO  [whvp::core] restored 35 pages
2020-01-30 16:39:36,879 INFO  [whvp::core] loading corpus
2020-01-30 16:39:36,879 INFO  [whvp::core] starting fuzzing
2020-01-30 16:39:36,881 INFO  [whvp::fuzz] 1 executions, 1 exec/s, coverage 2955, new 42,
     code 94.21 kB, data 229.38 kB, corpus 2, worklist 0, crashes 0
2020-01-30 16:39:37,882 INFO  [whvp::fuzz] 2017 executions, 2016 exec/s, coverage 2993,
     new 38, code 94.21 kB, data 229.38 kB, corpus 7, worklist 1, crashes 0
2020-01-30 16:39:38,883 INFO  [whvp::fuzz] 4137 executions, 2120 exec/s, coverage 2993,
     new 0, code 94.21 kB, data 229.38 kB, corpus 7, worklist 2, crashes 0
2020-01-30 16:39:39,883 INFO  [whvp::fuzz] 6098 executions, 1961 exec/s, coverage 3031,
     new 38, code 94.21 kB, data 229.38 kB, corpus 8, worklist 7, crashes 0
2020-01-30 16:39:40,883 INFO  [whvp::fuzz] 8440 executions, 2342 exec/s, coverage 3033,
     new 2, code 94.21 kB, data 229.38 kB, corpus 8, worklist 1, crashes 0
2020-01-30 16:39:41,344 INFO  [whvp] got abnormal exit on nt!KeBugCheckEx+0x0
     fffff80578dc48a0
2020-01-30 16:39:41,350 INFO  [whvp] saved 2 symbol(s)
2020-01-30 16:39:41,351 INFO  [whvp] saved 2 module(s)
2020-01-30 16:39:41,356 DEBUG [whvp::whvp] destructing partition
2020-01-30 16:39:41,360 DEBUG [whvp::mem] destructing allocator
```

Le buffer ayant provoqué le crash est le suivant :

```
$ hexdump -C fuzz/947f7fd2-3183-4b9b-87d3-df3b7815e210/crashes/nt_KeBugCheckEx+0
     x0_20200130_163941.bin
00000000  03 00 00 00 00 00 00 00  73 64 62 66 02 78 a2 38  |........sdbf.x.8|
00000010  01 60 01 98 00 00 00 00  03 78 0e 00 00 00 02 38  |.`.......x.....8|
00000020  07 70 03 38 04 90 01 98  00 00 00 00 03 78 20 38  |.p.8.........x 8|
00000030  01 60 01 98 02 38 17 70  03 38 01 60 01 98 00 00  |.`...8.p.8.`....|
00000040  00 00 03 78 0e 00 00 00  02 38 07 70 03 38 04 90  |...x.....8.p.8..|
00000050  01 98 00 00 00 00 03 78  20 00 00 00              |.......x ...|
```

Plusieurs octets ont été modifiés, l'étape suivante est de faire une analyse de la *root cause* afin de déterminer si ce crash est exploitable ou pas (hint : il ne l'est pas).

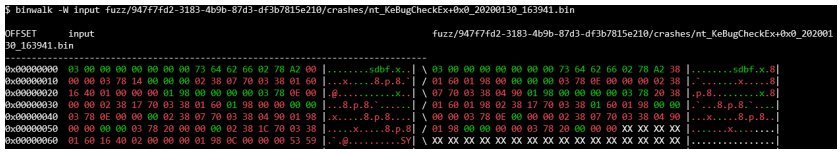

Fig. 4. Diff

5 Conclusion

Pour les lecteurs intéressés, le code utilisé pour les exemples est disponible sur `https://github.com/quarkslab/whvp`.

L'outil fonctionne, il y a néanmoins quelques limitations. Cette démarche a du sens uniquement sur du code synchrone. En effet un seul CPU est actif dans le guest et aucune interruption ne permet de le préempter. Par exemple si un accès à de la mémoire présente dans le swap

est effectué, aucun thread noyau n'est actif pour aller chercher cette page et la mapper en mémoire.

Il est bien sûr plus efficace d'exécuter du code relativement contraint. Si la fonction étudiée utilise l'intégralité du noyau, la partition va consommer beaucoup de mémoire et les performances seront affectées, la restauration des pages mémoire modifiées va prendre plus de temps.

Afin de juger des performances de WHVP, nous avons utilisé bochs à la place de WHVP. Bochs est plus rapide et plus précis pour obtenir une trace avec l'intégralité du contexte processeur (avec environ un facteur x10). Cela est dû au fait que chaque vm-exit est très coûteuse en temps. De multiples copies et validations de données sont effectuées à travers l'hyperviseur, le noyau et finalement l'espace utilisateur. Par contre pour du fuzzing les exécutions qui ne provoquent pas de vm-exits sont plus rapides car exécutées directement sur le processeur. Bochs permet en revanche d'instrumenter les comparaisons mémoire (ce qui est très utile pour du fuzzing).

Nous avons aussi fait un POC sur l'utilisation de notre DBI (QBDI) pour jitter le code exécuté dans la VM. L'idée est de se servir du moteur de la DBI pour modifier et instrumenter le code exécuté dans la vm. Par exemple on peut imaginer casser les comparaisons (*comparison unrolling*) pour faciliter le travail du fuzzer.

Il est à noter que cette démarche n'est pas lié à WHVP, il s'agit juste d'un hyperviseur parmi d'autres, rien n'empêche d'effectuer la même chose avec d'autres hyperviseurs (comme nous l'avons fait avec bochs).

Afin de faciliter l'écriture des tests, nous envisageons aussi de fournir la possibilité de décrire le prototype de la fonction étudiée et les entrées manipulées par le fuzzer avec par exemple un DSL dédié. Dans ce cas l'usage se rapprocherait très fortement de celui de *libfuzzer* avec le choix de notre fonction cible et l'écriture du harnais associé.

Le fuzzer décrit dans cet article est très simple, il a juste l'avantage d'exécuter très rapidement une fonction particulière. Avoir accès aux traces d'exécution permet d'envisager de le coupler avec d'autres outils comme *Triton* [14].

En annotant un buffer comme étant symbolique, on peut demander à *Triton* de nous fournir d'autres entrées qui exerceront de nouveaux chemins. Cette génération de nouvelles entrées est lente mais offre la garantie d'avoir une nouvelle entrée qui améliorera la couverture de code. Faire travailler de concert ces 2 générateurs d'entrées améliore la performance globale (en terme de couverture de code) du fuzzer.

Références

1. AFL. https://github.com/google/AFL.
2. applepie, a hypervisor implementation for Bochs. https://github.com/gamozolabs/applepie.
3. Combining Coverage-Guided and Generation-Based Fuzzing. https://fitzgeraldnick.com/2019/09/04/combining-coverage-guided-and-generation-based-fuzzing.html.
4. Fuzzing the Windows Kernel. https://github.com/yoava333/presentations/blob/master/Fuzzing%20the%20Windows%20Kernel%20-%20OffensiveCon%202020.pdf.
5. Honggfuzz. https://github.com/google/honggfuzz.
6. Lain. https://github.com/microsoft/lain.
7. LibFuzzer. https://llvm.org/docs/LibFuzzer.html.
8. pykd. https://githomelab.ru/pykd/pykd.
9. QBDI. https://github.com/QBDI/QBDI.
10. rpyc. https://github.com/tomerfiliba/rpyc.
11. Setting Up Long Mode. https://wiki.osdev.org/Setting_Up_Long_Mode.
12. Simpleator. https://github.com/ionescu007/Simpleator.
13. Structure Aware Fuzzing. https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md.
14. Triton. https://triton.quarkslab.com/.
15. Geoff Chappell. Kernel Shim Engine. http://www.geoffchappell.com/studies/windows/km/ntoskrnl/api/kshim/index.htm, 2016.
16. Jonathan Metzman. Going Beyond Coverage-Guided Fuzzing with Structured Fuzzing. https://i.blackhat.com/USA-19/Wednesday/us-19-Metzman-Going-Beyond-Coverage-Guided-Fuzzing-With-Structured-Fuzzing.pdf.
17. Microsoft. WhvCreatePartition. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvcreatepartition, 2017.
18. Microsoft. WhvCreateVirtualProcessor. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvcreatevirtualprocessor, 2017.
19. Microsoft. WhvGetCapability. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvgetcapability, 2017.

20. Microsoft. WHvGetVirtualProcessorRegisters. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvgetvirtualprocessorregisters, 2017.
21. Microsoft. WHvMapGpaRange. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvmapgparange, 2017.
22. Microsoft. WHvRunVirtualProcessor. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvrunvirtualprocessor, 2017.
23. Microsoft. WhvSetPartitionProperty. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvsetpartitionproperty, 2017.
24. Microsoft. WhvSetupPartition. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvsetuppartition, 2017.
25. Microsoft. Windows Hypervisor Platform. https://docs.microsoft.com/en-us/virtualization/api/, 2017.
26. Gabrielle Viala. Kernel Shim Engine for fun and profit. https://www.blackhoodie.re/assets/archive/Kernel_Shim_Engine_for_fun_-_pwissenlit.pdf, 2018.