Fuzz and Profit with WHVP

Damien Aumaitre, SSTIC 2020

Quarkslab

www.quarkslab.com

Motivation

- Toujours été intéressé par la recherche de vulnérabilités dans les composants du noyau Windows
- Commence par une exploration du code avec un désassembleur et un déboggueur (IDA et Windbg)
- L'habitude aidant on tombe sur une fonction qui a l'air intéressante mais trop complexe à regarder manuellement
- Naturellement on a envie du faire du fuzzing et là les ennuis commencent ...

Fuzzing kernel

- Pour faire du fuzzing kernel :
  - Habituellement on installe une VM (ou plusieurs) et le logiciel cible
  - On se débrouille pour envoyer nos données malformées
  - On utilise un déboggueur noyau pour monitorer notre VM
    - Le cycle envoi du test - restauration du snapshot peut être lent
    - Le déboggueur peut se désynchroniser
    - Automatiser le tout est pénible et demande pas mal de ressources

Fuzzing kernel

- Des projets existent pour pouvoir utiliser AFL sur du code noyau (kAFL)
- On peut aussi citer *syzcaller* (voir cette présentation pour une utilisation sous Windows)
- Approches globales qui ciblent les points d'entrées naturels du noyau (comme les appels systèmes)
- Le noyau est vaste et on se retrouve vite à tout tracer
  - Le nombre d'exécutions par seconde du fuzzer diminue
- Plus intéressé par une approche locale type fuzzing d'API (comme le fait *libfuzzer*)

WHVP

- En avril 2018, Microsoft met à disposition une nouvelle API appelé WHVP (Windows Hypervisor Platform)
  - Elle permet de créer des partitions Hyper-V avec un ou plusieurs processeurs virtuels
  - Elle permet de réagir aux événements se produisant dans la partition
- API introduite pour fournir la possibilité de faire fonctionner d'autres solutions de virtualisation au dessus d'Hyper-V (comme VMWare ou VirtualBox) sans devoir le désactiver
- Plusieurs outils utilisant cette API sont sortis, on peut citer par exemple :
  - `simpleator` qui permet de sandboxer un binaire userland
  - `applepie` qui permet d'utiliser Hyper-V dans bochs comme accélérateur

- Motivation : découvrir cette API et au passage appliquer ce qui se fait en fuzzing userland dans un contexte kernel
- Idée : faire du *coverage-guided snapshot-based fuzzing*
  - *snapshot-based* : avant l'exécution d'un testcase du fuzzer, on restaure l'état de la cible (cpu et mémoire)
  - *coveraged-guided* : utilisation de la couverture de code pour guider le fuzzer
  - Plutôt que de restaurer le snapshot d'une VM complète, restaurer le snapshot d'une vm minimaliste (plus rapide)

Cette présentation explique comment j'ai réalisé ce fuzzer (appliqué à WHVP mais tout à fait valable sur d'autres hyperviseurs)

Plusieurs questions se posent :

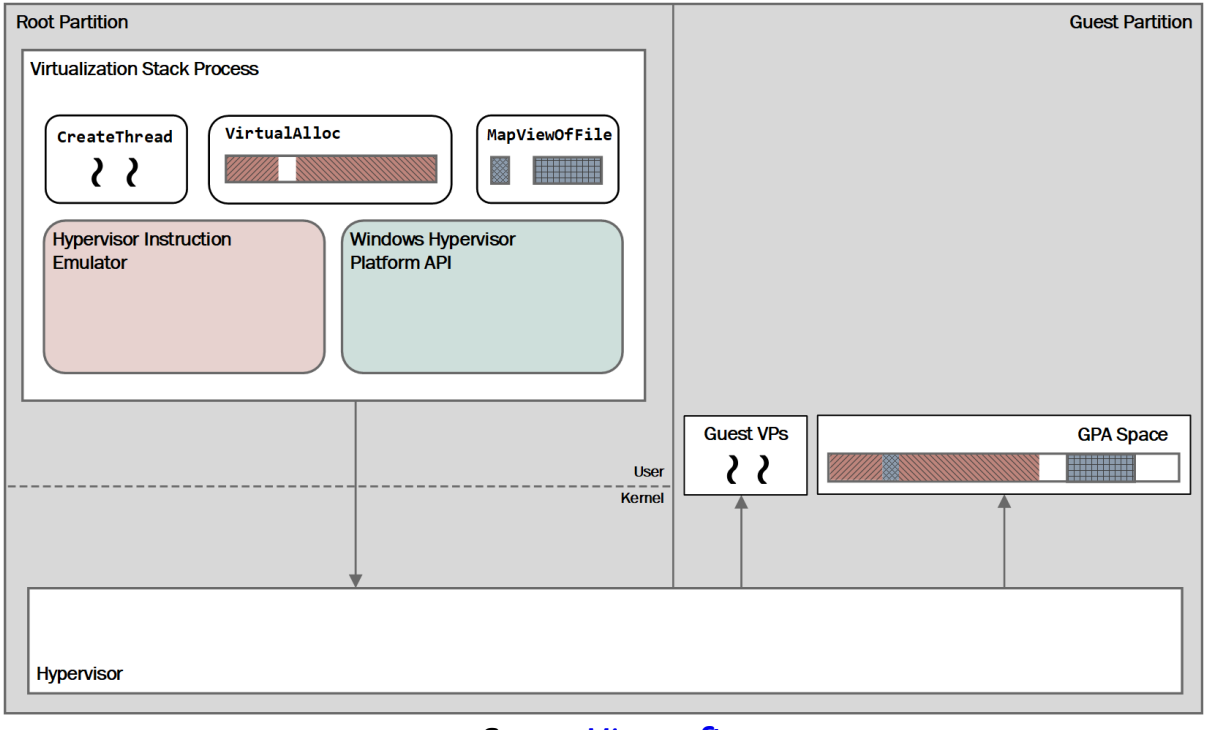
- Comment obtenir le snapshot de départ ?
- Comment obtenir cette vm minimaliste ?
- Comment récupérer la couverture de code ?

Snapshot

- Pour récupérer le snapshot de départ, j'ai utilisé un déboggueur noyau
- Facile pour l'état du processeur
- Pour la mémoire, plusieurs possibilités :
  - Dynamiquement en interrogeant un déboggueur connecté sur la cible
  - Statiquement en lisant un dump mémoire réalisé avec Windbg
  - Ou utiliser un snapshot mémoire d'une vm déjà existante

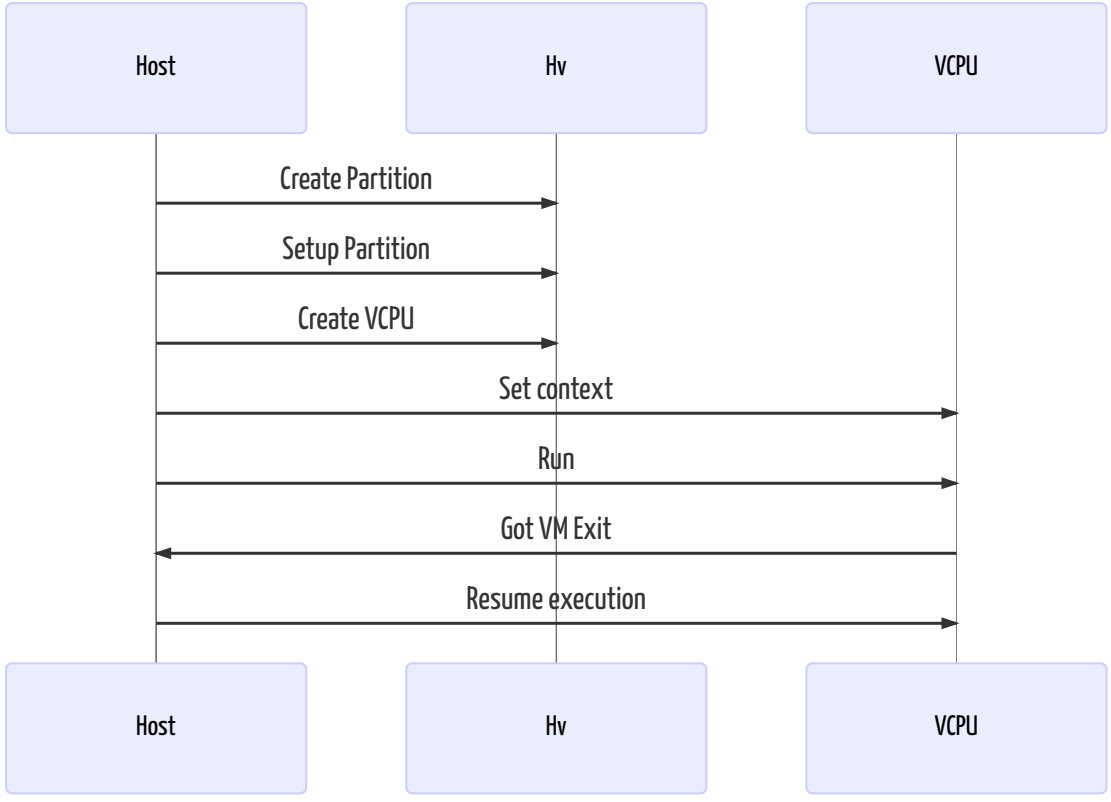
Snapshot

- Comment fournir ce snapshot à Hyper-V ?
- Mapper toute la mémoire revient à restaurer le snapshot d'une vm, pas très intéressant
- On va plutôt utilisé WHVP pour ça



Source [Microsoft]

WHVP


Host
Hv
VCPU
Create Partition
Setup Partition
Create VCPU
Set context
Run
Got VM Exit
Resume execution
Host
Hv
VCPU

Mode 64 bits

Premier problème :

- Le CPU virtuel démarre en mode 16 bits

Pour démarrer directement en mode 64 bits, on a 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

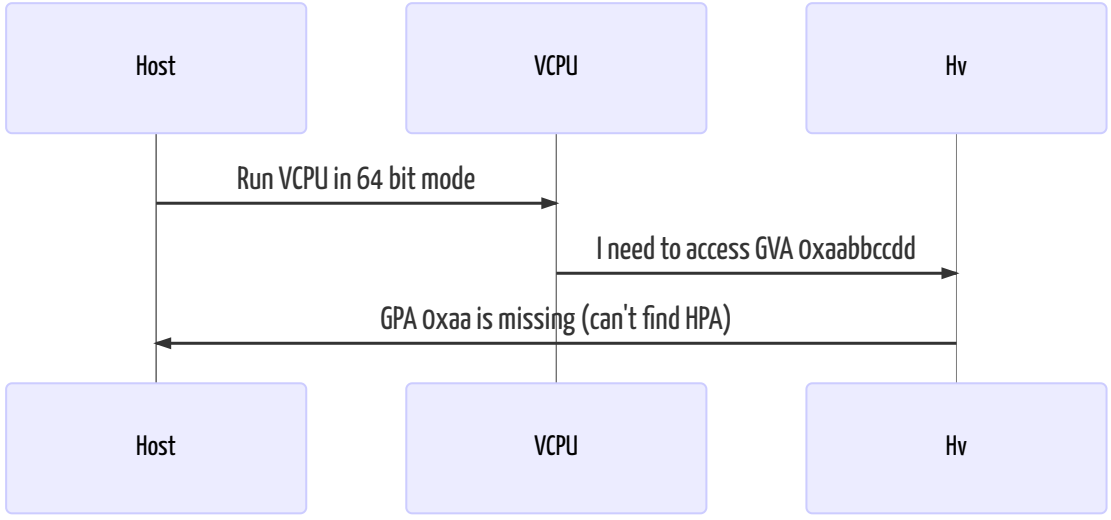
Plutôt que de créer *from scratch* un contexte valide, on va utiliser le contexte du snapshot

Pagination

Deuxième problème :

- La partition manipule des GPA (*Guest Physical Address*)
- La pagination est activée pour notre processeur virtuel
- Du coup on manipule des GVA (*Guest Virtual Address*)
- Comment mapper le code et les données du snapshot ?

Que se passe-t-il si une page physique manque lors de la résolution d'une adresse dans la partition ?



Pagination

- On peut donc mapper les GPA manquantes à la volée
- Il suffit d'interroger le snapshot pour savoir quelles sont les données présentes à l'adresse physique manquante
- Puis de mapper dans la partition les données

Exemple

```
kd> uf @rip
nt!RtlInitUnicodeString:
fffff80578c45b50 48c70100000000  mov     qword ptr [rcx],
fffff80578c45b57 48895108        mov     qword ptr [rcx+8
fffff80578c45b5b 4885d2          test    rdx,rdx
fffff80578c45b5e 7501            jne     nt!RtlInitUnicod
```

Exemple

```
kd> uf @rip
nt!RtlInitUnicodeString:
fffff80578c45b50 48c70100000000  mov     qword ptr [rcx]
fffff80578c45b57 48895108        mov     qword ptr [rcx+8
fffff80578c45b5b 4885d2          test    rdx,rdx
fffff80578c45b5e 7501            jne     nt!RtlInitUnicod
```

```
cr3 = 58147002
```

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

Exemple

```
kd> uf @rip
nt!RtlInitUnicodeString:
fffff80578c45b50 48c70100000000  mov     qword ptr [rcx]
fffff80578c45b57 48895108        mov     qword ptr [rcx+8
fffff80578c45b5b 4885d2          test    rdx,rdx
fffff80578c45b5e 7501            jne     nt!RtlInitUnicod
```

```
cr3 = 58147002

kd> !pte @rip
VA fffff80578c45b50

PXE at FFFFFC7E3F1F8F80
contains 0000000001108063
pfn 1108      ---DA--KWEV
```

```
MemoryAccess(
    VpContext {
        Rip: fffff80578c45b50,
        Rflags: 0000000000050246,
    },
    MemoryAccessContext {
        AccessInfo: MemoryAccessInfo {
            AccessType: Write,
            GpaUnmapped: true,
            GvaValid: false,
        },
        Gpa: 0000000011080a8,
        Gva: 0000000000000000,
    },
)
```

Exemple

```
kd> uf @rip
nt!RtlInitUnicodeString:
fffff80578c45b50 48c70100000000  mov     qword ptr [rcx]
fffff80578c45b57 48895108        mov     qword ptr [rcx+8
fffff80578c45b5b 4885d2          test    rdx,rdx
fffff80578c45b5e 7501            jne     nt!RtlInitUnicod
```

```
cr3 = 58147002

kd> !pte @rip
VA fffff80578c45b50

PXE at FFFFFC7E3F1F8F80
contains 0000000001108063
pfn 1108      ---DA--KWEV

PPE at FFFFFC7E3F1F00A8
contains 0000000001109063
pfn 1109      ---DA--KWEV
```

```
MemoryAccess(
    VpContext {
        Rip: fffff80578c45b50,
        Rflags: 0000000000050246,
    },
    MemoryAccessContext {
        AccessInfo: MemoryAccessInfo {
            AccessType: Write,
            GpaUnmapped: true,
            GvaValid: false,
        },
        Gpa: 000000001109e30,
        Gva: 0000000000000000,
    },
)
```

Exemple

```
kd> uf @rip
nt!RtlInitUnicodeString:
fffff80578c45b50 48c70100000000  mov     qword ptr [rcx]
fffff80578c45b57 48895108        mov     qword ptr [rcx+8
fffff80578c45b5b 4885d2          test    rdx,rdx
fffff80578c45b5e 7501            jne     nt!RtlInitUnicod
```

```
cr3 = 58147002

kd> !pte @rip
VA fffff80578c45b50

PXE at FFFFFC7E3F1F8F80
contains 0000000001108063
pfn 1108      ---DA--KWEV

PPE at FFFFFC7E3F1F00A8
contains 0000000001109063
pfn 1109      ---DA--KWEV

PDE at FFFFFC7E3E015E30
contains 0000000001113063
pfn 1113      ---DA--KWEV
```

```
MemoryAccess(
    VpContext {
        Rip: fffff80578c45b50,
        Rflags: 0000000000050246,
    },
    MemoryAccessContext {
        AccessInfo: MemoryAccessInfo {
            AccessType: Write,
            GpaUnmapped: true,
            GvaValid: false,
        },
        Gpa: 000000001113228,
        Gva: 0000000000000000,
    },
)
```

Exemple

```
kd> uf @rip
nt!RtlInitUnicodeString:
fffff80578c45b50 48c70100000000  mov     qword ptr [rcx]
fffff80578c45b57 48895108        mov     qword ptr [rcx+8
fffff80578c45b5b 4885d2          test    rdx,rdx
fffff80578c45b5e 7501            jne     nt!RtlInitUnicod
```

```
cr3 = 58147002

kd> !pte @rip
VA fffff80578c45b50

PXE at FFFFFC7E3F1F8F80
contains 0000000001108063
pfn 1108      ---DA--KWEV

PPE at FFFFFC7E3F1F00A8
contains 0000000001109063
pfn 1109      ---DA--KWEV

PDE at FFFFFC7E3E015E30
contains 0000000001113063
pfn 1113      ---DA--KWEV

PTE at FFFFFC7C02BC6228
contains 0900000001F3D021
pfn 1f3d      ----A--KREV
```

```
MemoryAccess(
      VpContext {
          Rip: fffff80578c45b50,
          Rflags: 0000000000050246,
      },
      MemoryAccessContext {
          AccessInfo: MemoryAccessInfo {
              AccessType: Execute,
              GpaUnmapped: true,
              GvaValid: true,
          },
          Gpa: 000000001f3db50,
          Gva: fffff80578c45b50,
      },
  )
```

Conditions d'arrêt

Troisième problème :

- On veut exécuter une fonction spécifique
- Si on continue l'exécution de notre processeur virtuel on va finir par mapper et exécuter l'intégralité du noyau
  - Pas vraiment ce que l'on cherche à faire
- Il nous faut des conditions d'arrêt :
  - Pour savoir quand notre fonction cible se termine
  - Pour savoir si on exécute du code *autre* (on pense naturellement à `KeBugCheck` et consorts)

Conditions d'arrêt

- Pour obtenir l'adresse de retour c'est facile, elle est dans la pile
- Pour les adresses correspondant au code *autre*, il suffit de résoudre les adresses grâce au déboggueur lors de la création du snapshot
- Il suffit ensuite d'insérer des points d'arrêts logiciels sur ces différentes adresses au moment où on mappe la mémoire dans la partition
- Et de configurer la partition pour être averti de ce type d'exception

Il en est où mon fuzzer ?

Récapitulatif :

- On peut démarrer un processeur virtuel avec un contexte arbitraire
- Le code est mappé à la volée dans la partition Hyper-V : on peut le modifier si besoin
- La mémoire de la partition ne contient que les pages nécessaires à l'exécution de notre fonction cible
- On peut restaurer la mémoire et le contexte processeur et exécuter notre fonction cible autant de fois que l'on veut
- La restauration du code est facilitée par la présence d'une fonction permettant de savoir quelles pages physiques ont été modifiées (et donc à restaurer)
- Tout est isolé et l'exécution est déterministe, ce sont des conditions parfaites pour du fuzzing

Par contre on n'a aucune idée de ce qu'on a exécuté (un peu embêtant pour faire du fuzzing assisté par couverture de code)

Couverture de code

Quatrième problème :

- Comment obtenir la couverture de code ?
- Solution de facilité : on active le trapflag (TF) dans le registre `rflags` pour passer en mode single step
- On obtient donc une sortie de VM pour chaque instruction exécutée
- Il suffit de récupérer le contexte processeur et de reprendre l'exécution ensuite
- Problème : chaque sortie de VM est coûteuse
  - Le temps d'exécution va être plus lent
  - Les performances pour faire du fuzzing vont en pâtir

Couverture de code

- On veut faire du fuzzing assisté par couverture de code
  - Ce qui nous intéresse, ce sont les **nouvelles** instructions exécutées
- Solution : on mappe les pages de code avec des `0xcc`
  - Chaque fois qu'une nouvelle instruction est exécutée on remplace les octets de l'instruction par l'instruction originale en lisant la mémoire du snapshot
  - Le temps d'exécution sera plus rapide sauf si on exécute une nouvelle branche

- Comment fonctionne un fuzzer assisté par de la couverture de code ?
  - La cible est instrumentée pour récupérer la couverture de code

Corpus

- Comment fonctionne un fuzzer assisté par de la couverture de code ?
  - ~~La cible est instrumentée pour récupérer la couverture de code~~
  - On charge un corpus d'entrées dans le fuzzer (pas obligatoire)

Corpus

- Analyse précise d'une fonction particulière
- Prototype de la fonction ainsi que les paramètres contrôlés par l'attaquant sont connus
- Entrées contrôlées par l'attaquant vues comme un buffer avec une adresse et une taille
- Les données (valides) originales sont lues de la mémoire de la partition après une première exécution

Mutation

- Comment fonctionne un fuzzer assisté par de la couverture de code ?
  - ~~La cible est instrumentée pour récupérer la couverture de code~~
  - ~~On charge un corpus d'entrées dans le fuzzer (pas obligatoire)~~
  - Une entrée du corpus est choisie et mutée

Mutation

- La stratégie pour sélectionner une entrée du corpus est minimaliste
- On applique les mutations à chaque entrée présente dans le corpus (qui ont donc des couvertures de code différentes)
- Pas de fuzzing structuré
- Heuristiques classiques de mutation similaires à celles utilisés dans radamsa ou AFL

Exécution

- Comment fonctionne un fuzzer assisté par de la couverture de code ?
  - ~~La cible est instrumentée pour récupérer la couverture de code~~
  - ~~On charge un corpus d'entrées dans le fuzzer (pas obligatoire)~~
  - ~~Une entrée du corpus est choisie et mutée~~
  - L'entrée choisie est exécutée

Exécution

- L'entrée mutée est copiée dans la mémoire de la partition
- Le contexte cpu et mémoire de la partition est restauré
- Si on obtient de la nouvelle couverture de code, on rajoute l'entrée au corpus
- Si la fonction est exécutée sans passer par l'adresse de retour attendue, on loggue l'entrée (qui est considéré comme un crash)

Triage

- 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
- Cela permet aussi d'obtenir les traces complètes d'exécution (qui aideront pour l'analyse de la root cause)

Démo

Conclusion

- Les performances sont intéressantes pour faire du fuzzing (environ 2000 exécutions par seconde dans la démo)
- Elles dépendent évidemment du code qui est ciblé (le nombre de VM exits et la restauration du snapshot sont coûteux)
- Adapté uniquement pour du code synchrone qui ne dépend pas d'un périphérique (pas d'OS ni de support hardware dans la VM)
- Parfait pour cibler des bouts de code précis
- Appliqué à WHVP mais pas obligatoire
  - Pour avoir une idée des performances de WHVP, on a appliqué la même démarche en utilisant Bochs (la partie instrumentation)
  - Bochs est plus rapide pour faire une trace complète mais plus lent pour faire du fuzzing car toutes les instructions sont émulées
- Pour les curieux et les aventureux, le code est disponible sur https://github.com/quarkslab/whvp/

Questions ?