Vous avez obtenu un trophée : PS4 jailbreaké

Quentin Meffre et Mehdi Talbi
quentin.meffre@synacktiv.com
mehdi.talbi@synacktiv.com

Synacktiv

Résumé. En dépit d'une communauté active sur le hacking de consoles de jeux vidéo, il existe que très peu de codes d'exploitation publics sur la PlayStation 4. Cet article détaille la stratégie que nous avons adoptée afin d'exploiter une vulnérabilité 0-day que nous avons identifiée dans le moteur WebKit sur lequel s'appuie le navigateur de la PS4.

1 Introduction

Le navigateur de la PlayStation 4 constitue sans doute la surface d'attaque la plus ciblée pour un jailbreak de la console. Cependant, les techniques de durcissement dont bénéficient les navigateurs actuels couplés à l'absence de capacité de débogage rendent difficile l'exploitation de bugs sur les derniers firmwares de la PS4.

Cet article détaille la stratégie d'exploitation que nous avons adoptée afin d'exploiter une vulnérabilité 0-day dans WebKit. Il s'agit d'une vulnérabilité de type Use-After-Free qui n'offre de prime abord que des primitives limitées. Cependant, grâce à une faiblesse identifiée dans l'ASLR, il a été possible d'exploiter cette vulnérabilité menant au premier jailbreak public sur la version 7 de la PS4.

Le présent article est structuré comme suit : la section 1.1 présente l'état de l'art. L'exploitation de la vulnérabilité nécessite une compréhension des rouages internes de l'allocateur standard de WebKit qui sera introduit en section 2. La vulnérabilité sera détaillée dans la section 3 et la stratégie de son exploitation sera présentée en section 4. Finalement, nous présenterons en section 5 nos conclusions et ce que nous avons planifié comme travaux futurs.

1.1 État de l'art

Le navigateur est le point d'entrée le plus commun pour attaquer la PS4. Le navigateur est basé sur WebKit et tourne dans une sandbox. Cependant, certaines contre-mesures telles que la GigaCage [10] ou bien

la randomisation des *StructureID* [14] sont absentes. Par ailleurs, le JIT est désactivé ce qui peut rendre plus difficile l'obtention de l'exécution de code dans le contexte du processus cible.

Une chaîne d'exploitation typique débute par un exploit WebKit permettant d'obtenir de l'exécution de code dans le contexte du processus responsable du rendu HTML, suivi d'un contournement de la sandbox afin de lancer un exploit kernel permettant d'élever ses privilèges sur la console.

Il y a eu par le passé quelques exploits WebKit. Le dernier en date est l'exploit "bad-hoist" [1] qui exploite la vulnérabilité CVE-2018-4386 identifié initialement par l'équipe de sécurité Projet Zero (P0). L'exploit "bad-hoist" cible les firmwares 6.xx et permet d'obtenir des accès en lecture/écriture à la mémoire du processus cible. Précédemment, la vulnérabilité CVE-2018-4441, également identifié par P0, a fait également l'objet d'une exploitation sur les firmwares 6.20. Pour les firmwares antérieurs à la version 6, quelques exploits sont également disponibles [2, 8].

Concernant les exploits kernel, le dernier en date exploite une vulnérabilité identifiée par Andy Nguyen [12] dans la pile protocolaire IPv6. Cette vulnérabilité a été utilisée conjointement avec la vulnérabilité introduite dans cet article suite à la publication de notre exploit pour constituer le premier jailbreak public sur la version 7.02 [6]. D'autres vulnérabilités ont été rendues publics récemment par le même auteur et sur lequel s'affairent plusieurs hackers afin de disposer d'une nouvelle chaîne sur les dernières versions 7.xx.

Finalement, quelques vulnérabilités dans BPF ont été également exploitées dans les versions antérieures à la version 6 [4, 5].

2 Les allocateurs WebKit

Webkit utilise plusieurs allocateurs dans sa base de code :

- *FastMalloc* est l'allocateur standard ;
- *IsoHeap* est utilisé par le moteur de rendu. Cet allocateur trie les allocations en fonction de leurs types dans le but de rendre difficile l'exploitation de vulnérabilités permettant de confondre deux objets ;
- Le *GC* (*Garbage Collector*) est utilisé par le moteur JavaScript pour allouer des objets JavaScript ;
- *IsoSubspace* est utilisé par le moteur JavaScript. Cet allocateur trie chaque allocation par taille mais, très peu d'objets sont alloués par *IsoSubspace* [11] ;

— *GigaCage* est une protection empêchant d'écrire ou de lire en dehors des limites de certains objets. Cette protection est désactivée par défaut sur la PS4.

2.1 Allocateur standard

L'allocateur standard est composé de chunks (*Chunk*) qui sont divisés en pages (*smallpages*) de 4 ko. Une page est à son tour divisée en lignes de 256 octets (*smalllines*) servant chacune des allocations de même taille. La figure 1 illustre la structuration du heap.

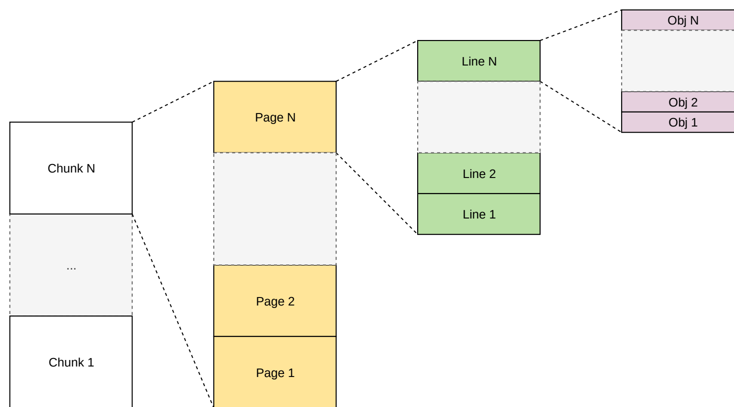

Fig. 1. Allocateur standard

L'allocateur standard est un allocateur de type "bump-pointer" où chaque allocation consiste à incrémenter un pointeur :

```
--m_remaining;
char* result = m_ptr;
m_ptr += m_size;
return result;
```

Les objets sont alloués via la primitive `fastMalloc` qui peut emprunter soit le chemin rapide consistant à exécuter le code illustré plus haut soit le chemin lent nécessitant de réapprovisionner l'allocateur au préalable.

Le réapprovisionnement de l'allocateur se fait à partir d'un cache dédié, dénommé `bumpRangeCache`. Lorsque celui-ci est également à court d'objets, une nouvelle page est allouée afin d'alimenter l'allocateur. La nouvelle page est soit retirée du cache `lineCache`, ou bien extraite dans le cas contraire, de la liste des pages libres maintenue pour chaque chunk. Dans le cas

d'une page fragmentée (i.e. page issue du cache `lineCache`), les lignes libres et contiguës de la page sont utilisées pour alimenter l'allocateur. Le reste des lignes disponibles sert à alimenter le cache `bumpRangeCache`. La figure 2 illustre le réapprovisionnement de l'allocateur.

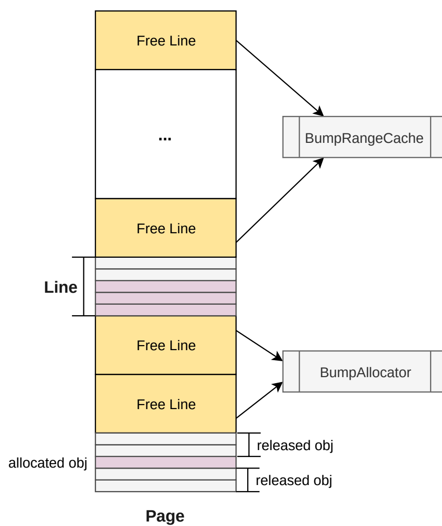

Fig. 2. Réapprovisionnement de l'allocateur

Lorsqu'un objet est libéré, il n'est pas mis immédiatement à disposition pour les futures allocations. Il est tout d'abord placé dans un vecteur dénommé `m_objectLog` et qui est manipulé dans la fonction `Deallocator::processObjectLog` lorsque celui-ci atteint sa capacité maximale (512 objets) ou lors du réapprovisionnement de l'allocateur. Le rôle de la fonction `Deallocator::processObjectLog` est de libérer des lignes lorsque celles-ci ne sont plus référencées (i.e. tous les objets contenus dans la ligne ont été libérés). Lorsqu'une ligne est libre, la page correspondante est insérée dans le cache `cacheLine`. Il est à noter qu'un compteur de références est associé aux lignes, aux pages et aux chunks. Ces éléments sont libérés lorsque le compteur de références atteint la valeur 0.

3 Le bug

La vulnérabilité a été remontée par le fuzzer interne de Synacktiv. Le problème vient de la fonction

`WebCore::ValidationMessage::buildBubbleTree`, utilisée par le moteur de rendu.

```
void ValidationMessage::buildBubbleTree()
{
    /* ... */
    auto weakElement = makeWeakPtr(*m_element);

    document.updateLayout();

    if (!weakElement || !m_element->renderer())
        return;

    adjustBubblePosition(m_element->renderer()->
        absoluteBoundingBoxRect(), m_bubble.get());

    /* ... */
}
```

Listing 1. Code vulnérable

Dans le listing 1, la fonction `updateLayout` est invoquée afin de mettre à jour la disposition de la page. Durant cet appel, les événements JavaScript enregistrés par l'utilisateur sont potentiellement exécutés. Si durant un tel événement (p. ex. focus sur un élément HTML), l'objet `ValidationMessage` est détruit, cela conduirait à une situation de type Use-After-Free au retour de la fonction vulnérable.

Les développeurs de WebKit ont identifié les fonctions permettant de mettre à jour le style et la disposition d'une page comme pouvant amener à des situations de Use-After-Free. En témoigne l'extrait suivant issu de la révision `r245823` [3] : "If a method decides a layout or style update is needed, it needs to confirm that the elements it was operating on are still valid and needed in the current operation". Le patch en question protège les attributs d'une classe avant des appels à la fonction `updateLayout`. Le but étant d'empêcher de libérer un objet lors d'un évènement JavaScript. Dans le cas de notre vulnérabilité 1, les développeurs ont ajouté un `WeakPtr` à partir de l'attribut `m_element` afin d'empêcher la libération de ce dernier. Cependant, la construction de ce `WeakPtr` est incorrecte. Afin d'être valide, un `WeakPtr` doit être construit à partir d'un pointeur et `m_element` est un pointeur. Cet attribut est déréférencé lorsqu'il est passé en paramètre à la fonction `makeWeakPtr`. Ce qui construit un `WeakPtr` à partir du contenu de `m_element` et non pas à partir du pointeur de ce dernier. Cette erreur implique qu'il est toujours possible de détruire l'objet `ValidationMessage` durant un événement JavaScript et obtenir une situation de Use-After-Free.

La vulnérabilité a été remontée de manière responsable aux développeurs de WebKit qui ont soumis le patch illustré par le listing 2 pour corriger la vulnérabilité.

```
void ValidationMessage::buildBubbleTree()
{
    /* ... */
-
-   auto weakElement = makeWeakPtr(*m_element);
-
-   document.updateLayout();
-
-   if (!weakElement || !m_element->renderer())
-       return;
-
-   adjustBubblePosition(m_element->renderer()->
    absoluteBoundingBoxRect(), m_bubble.get());

    /* ... */


+   if (!document.view())
+       return;
+   document.view()->queuePostLayoutCallback([weakThis = makeWeakPtr
    (*this)] {
+       if (!weakThis)
+           return;
+       weakThis->adjustBubblePosition();
+   });
}
```

Listing 2. Correctif

3.1 Le chemin vulnérable

La figure 3 suivante illustre le chemin vulnérable. L'objet vulnérable `ValidationMessage` peut-être instancié en invoquant la méthode `reportValidity` sur le champ d'un formulaire HTML. Il est possible maintenant d'atteindre le chemin vulnérable en enregistrant un événement JS sur ce champ HTML (e.g. `onfocus`).

Lorsqu'elle est appelée, la méthode `reportValidity` déclenche un minuteur afin d'invoquer la fonction vulnérable `buildBubbleTree`. Si le curseur est positionné sur le champ HTML cible avant l'expiration du minuteur, la callback JavaScript associée à cet événement sera exécuté. Si durant cette callback, l'objet `ValidationMessage` est détruit, cela aboutirait à une vulnérabilité de type Use-After-Free.

Maintenant, si d'une certaine manière nous parvenons à survivre au crash dû aux accès invalides à certains champs de l'objet

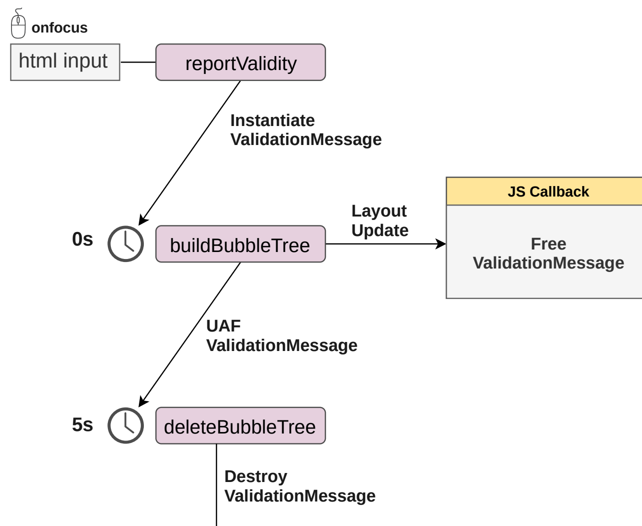

Fig. 3. Chemin vulnérable

`ValidationMessage` lors du retour dans la fonction `buildBubbleTree`, nous atteindrons la fonction `deleteBubbleTree` qui aura pour conséquence de détruire à nouveau l'instance `ValidationMessage`.

Notre première tentative pour déclencher la vulnérabilité a échoué. La raison est due à la méthode `reportValidity` qui positionne le focus sur l'élément HTML cible, ce qui a pour effet de déclencher l'exécution de l'événement JavaScript prématurément. Il est possible de contourner ce problème en ayant par exemple recours à deux champs de texte HTML : `input1` et `input2`. Tout d'abord, nous enregistrons un événement JavaScript sur le premier champ qui va simplement mettre le curseur sur le second champ HTML lorsque cet événement sera déclenché par la méthode `reportValidity`. Ensuite, avant l'expiration du minuteur, nous redéfinissons la callback JS sur le premier élément `input1` afin de détruire l'instance de l'objet `ValidationMessage`. Ce scénario est illustré par la figure 4.

3.2 Débogage du bug

Le déclenchement du bug sur la PS4 résulte en un crash et un redémarrage du navigateur. En l'absence d'information de débogage, deux options sont possibles pour l'exploitation de cette vulnérabilité sur la PS4 :

— Mettre en place un environnement de travail qui soit le plus proche possible de celui de la console. Cela consiste à installer une dis-

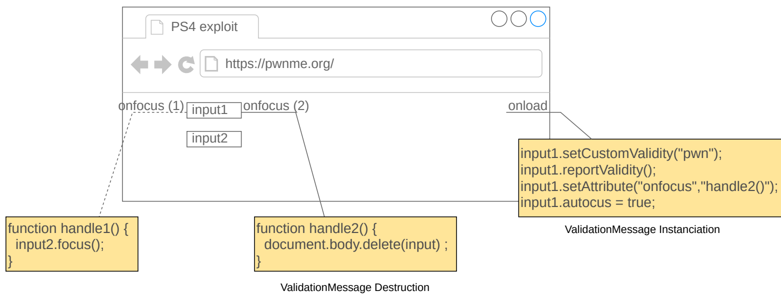

Fig. 4. Déclenchement de la vulnérabilité

tribution FreeBSD sur laquelle seront compilées les sources de WebKit récupérées depuis le site de Sony [9]. Cette option est utile, mais malheureusement un code d'exploitation fonctionnel sur notre environnement n'implique pas un portage assuré sur la console ;

— Déboguer une vulnérabilité 0-day en utilisant un code d'exploitation d'une vulnérabilité 1-day. L'exploit "bad-hoist" permet de répondre à cette problématique étant donné qu'il est doté de primitives de lecture/écriture, mais également des primitives classiques addrof/fakeobj. Cet exploit ne fonctionne malheureusement qu'en versions 6 de la PS4 et malgré sa faible fiabilité, c'est cette option qui a été retenue durant la phase d'exploitation. Il est à noter finalement que lancer l'exploit "bad-hoist" peut parasiter notre stratégie pour façonner le tas.

3.3 Anatomie d'un objet vulnérable

L'objet vulnérable `ValidationMessage` est instancié par la fonction `reportValidity`, et est principalement accédé par la méthode `buildBubbleTree`. L'objet est alloué via `fastMalloc`. Il est constitué des champs illustrés par la figure 5. Certains champs de classe sont instanciés (`m_messageBody`, `m_messageHeading`) et/ou ré-instanciés (`m_timer`) après une mise à jour de la disposition (p. ex. après l'appel à `updateLayout`). Les champs `m_bubble` et `m_element` sont accédés quant à eux après une mise à jour de la disposition. Ils pointent chacun sur une instance d'un objet `HTMLElement`.

L'instance `ValidationMessage` est détruite par la méthode `deleteBubbleTree` :

```
void ValidationMessage::deleteBubbleTree()
{
```

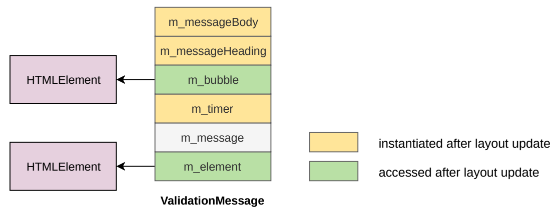

Fig. 5. Objet ValidationMessage

```
    if (m_bubble) {
        m_messageHeading = nullptr;
        m_messageBody = nullptr;
        m_element->userAgentShadowRoot()->removeChild(*m_bubble);
        m_bubble = nullptr;
    }
    m_message = String();
}
```

La méthode `deleteBubbleTree` assigne un pointeur à la plupart des champs de l'objet `ValidationMessage` provoquant un crash du navigateur lors du déréférencement du champ `m_bubble` dans la fonction `buildBubbleTree`.

3.4 Survivre au crash initial

Afin d'exploiter cette vulnérabilité, nous devons disposer soit d'une fuite de la mémoire du processus cible ou bien d'un moyen permettant de contourner l'ASLR. Il se trouve qu'en allouant certains types d'objets plusieurs milliers de fois, ces derniers finissent par être localisés à des adresses prédictibles. Ce comportement a pu être observé grâce à l'exploit "bad-hoist" qui nous a permis d'identifier une adresse à laquelle on peut forcer l'allocation d'un objet de type `HTMLElement` en version 6 de la PS4. En section 4.8, nous décrivons une procédure permettant de bruteforcer cette adresse en version 7.

Disposant désormais d'un moyen de contourner l'ASLR, il est possible d'éviter le crash initial en procédant comme suit (voir figure 6) :

1. Forcer l'allocation d'un objet `HTMLElement` à une adresse prédictible via l'allocation massive d'objets `HTMLTextAreaElement` ;
2. Exploiter l'UAF et confondre l'objet `ValidationMessage` avec un objet contrôlé (contenu d'un `ArrayBuffer`) ;
3. Ajuster les valeurs des champs `m_bubble` et `m_element` de telle sorte à ce qu'elles pointent sur l'adresse prédite et dans laquelle réside une instance `HTMLTextAreaElement`.

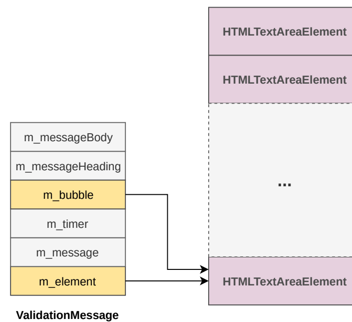

Fig. 6. Exploitation UAF

4 Stratégie d'exploitation

4.1 Ré-utiliser l'objet cible

Afin de ré-utiliser l'objet `ValidationMessage` libéré, nous avons suivi les étapes suivantes, telles que décrites sur la figure 7.

1. Nous allouons beaucoup d'objets O sur le tas. L'objet O doit faire la même taille que l'objet `ValidationMessage` (48 octets). Ces objets O vont être alloués avant et après l'allocation de l'objet `ValidationMessage` ciblé ;
2. Nous libérons l'objet `ValidationMessage` ciblé ainsi que les objets O alloués autour de notre objet cible. Cela va permettre de libérer la `SmallLine` contenant l'objet cible. La `SmallPage` associée va être mise en cache par l'allocateur `FastMalloc` ;
3. Nous allouons à nouveau beaucoup d'objets ayant la même taille que notre objet cible. Dans notre exploit, nous utilisons le tampon de données alloué par l'objet `ArrayBuffer` dans le but d'avoir deux références différentes qui pointent sur la même zone mémoire.

4.2 Fuite de mémoire initiale

Comme cité précédemment en section 3.3, certains attributs de l'objet `ValidationMessage` sont instanciés après la mise à jour de la disposition d'une page. Il est possible d'obtenir leurs valeurs en lisant le contenu de l'`ArrayBuffer`. Plus précisément, il est possible d'obtenir les valeurs des attributs `m_messageBody`, `m_messageHeading` et `m_timer`. Le champ de classe `m_timer` est particulièrement intéressant puisqu'il s'agit d'un objet alloué par l'allocateur `FastMalloc`. Cette fuite d'information sera utilisée plus tard afin d'inférer l'adresse d'objets alloués sur la même `SmallPage`.

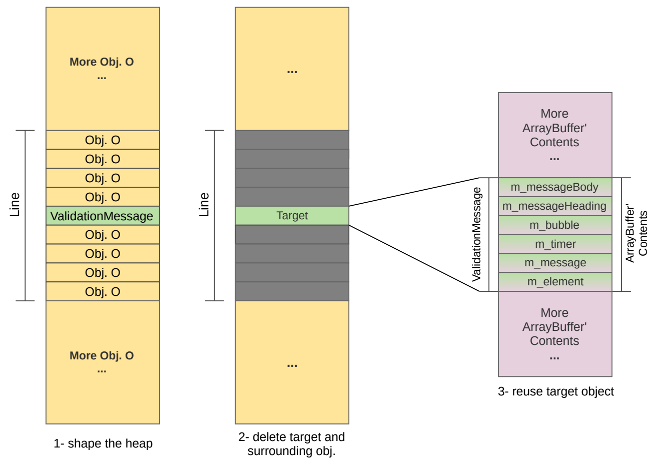

Fig. 7. Disposition des allocations faite autour de l'objet `ValidationMessage`.

4.3 La primitive de décrément arbitraire

Si les champs de l'objet cible `ValidationMessage` sont restaurés correctement comme décrit par la figure 6, la méthode `deleteBubbleTree` sera appelée après l'expiration d'un minuteur. Cette méthode affecte la valeur `NULL` à certains attributs. Il est important de noter que l'affectation de la valeur `NULL` sur un objet de la famille des `RefPtr` est surchargée par une méthode qui a pour but de décrémenter un compteur de références présent dans l'objet alloué. Cela signifie que nous sommes capables de décrémenter un compteur de références sur plusieurs attributs contrôlés de l'objet `ValidationMessage` : `m_messageBody`, `m_messageHeading` et `m_bubble`.

Le décrément du compteur de références est exploitable en altérant le pointeur de l'attribut `m_messageHeading`. Il est alors possible de confondre le compteur de références avec l'attribut responsable de la taille d'un autre objet. Par exemple, l'objet `StringImpl` possède un attribut `length` ainsi que des données. Le décrément arbitraire peut nous permettre de corrompre l'attribut `length` afin d'avoir un objet `StringImpl` possédant une taille plus grande que le tampon de données alloué et ainsi nous pourrions lire au-delà des limites de ce tampon. La figure 8 illustre ce scénario.

Notre exploit utilise deux fois la primitive de décrément arbitraire :

1. La première étape, détaillée en section 4.4, consiste à obtenir une primitive de lecture relative afin d'obtenir l'adresse d'un objet de type `JSArrayBufferView` ;

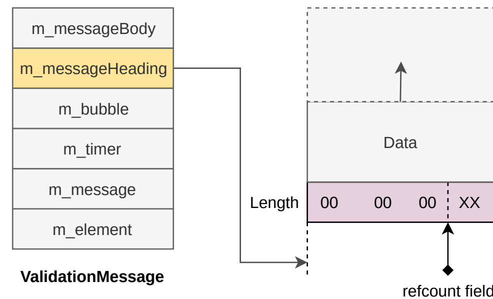

Fig. 8. Corruption de la taille d'un objet `StringImpl`.

2. La seconde étape, détaillée en section 4.5, consiste à obtenir une primitive de lecture/écriture relative en corrompant l'attribut `length` de l'objet `JSArrayBufferView` dont la référence est obtenue lors de la première étape.

La figure 9 illustre les étapes principales de notre exploit.

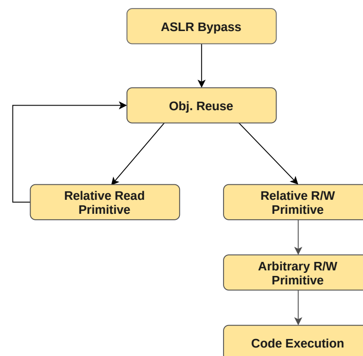

Fig. 9. Étape permettant d'obtenir une lecture et écriture arbitraire.

4.4 La primitive de lecture relative

Le but de cette partie est d'obtenir l'adresse d'un objet `JSArrayBufferView` alloué. Cet objet est intéressant dans notre contexte puisqu'il a un attribut `length` et il permet de lire et écrire des données arbitraires dans un tampon. Si nous contrôlons l'attribut `length` de manière à le rendre plus grand que sa valeur initiale, alors nous pourrons lire et écrire au-delà des limites du tampon de données. Cet objet est alloué par

le GC (*Garbage Collector*). Cet allocateur utilise un tas différent de celui utilisé par `FastMalloc` ce qui signifie que nous ne pouvons pas corrompre des objets alloués par le GC pour l'instant.

L'objet `StringImpl` est un objet utilisé pour représenter une chaîne de caractères dans le moteur JavaScript de Webkit. Cet objet a un attribut `length` et il permet de lire des données dans un tampon. En JavaScript les chaînes de caractères sont immuables, cela signifie que nous pouvons lire, mais pas écrire dans le tampon après l'avoir initialisé. L'objet `StringImpl` est alloué par `FastMalloc` et la taille de l'objet est partiellement contrôlable. Cet objet est une bonne cible pour obtenir une primitive de lecture relative.

Voici la stratégie utilisée pour corrompre l'attribut `length` d'un objet `StringImpl` alloué :

1. Allouer beaucoup d'objets `StringImpl` avant et après l'objet `m_timer` dont nous possédons l'adresse. Les objets `StringImpl` alloués font la même taille qu'un objet de type `Timer` ;
2. Utiliser la primitive de décrément arbitraire afin de corrompre l'attribut `length` de l'un des objets `StringImpl` alloué ;
3. Itérer sur chaque objet `StringImpl` alloué afin de trouver celui qui a été corrompu. L'objet qui a une taille excessivement grande est l'objet corrompu.

La figure 10 illustre la représentation mémoire attendue.

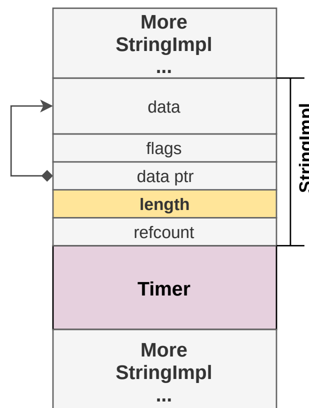

Fig. 10. Représentation mémoire du tas après avoir alloué les objets `StringImpl`.

À partir de cette nouvelle primitive permettant de lire des valeurs se trouvant dans le tas au-delà du tampon de l'objet `StringImpl`, nous allons voir comment retrouver l'adresse d'un objet `JSArrayBufferView`.

La méthode `Object.defineProperties` est une méthode native en JavaScript permettant de définir plusieurs propriétés à un objet. Le listing 3 illustre l'implémentation de la méthode `defineProperties` faite par WebKit.

```
static JSValue defineProperties(ExecState* exec, JSObject*
    object, JSObject* properties)
{
    Vector<PropertyDescriptor> descriptors;
    MarkedArgumentBuffer markBuffer;

    /* ... */
    JSValue prop = properties->get(exec, propertyNames[i]);
    /* ... */
    PropertyDescriptor descriptor;
    toPropertyDescriptor(exec, prop, descriptor);
    /* ... */
    descriptors.append(descriptor);        // [1] store JSValue
        reference on fastMalloc
    /* ... */
    markBuffer.append(descriptor.value()); // [2] store one more
         JSValue reference on fastMalloc
}
```

Listing 3. Implémentation de la méthode `Object.defineProperties`

Cette implémentation utilise deux objets intéressants d'un point de vu exploitabilité :

— `Vector<PropertyDescriptor>` ;

— `MarkedArgumentBuffer`.

Ces deux objets possèdent un tampon alloué par l'allocateur `FastMalloc` et ces deux objets sont utilisés par la méthode `defineProperties` pour stocker des `JSValue` dans leurs tampons. La classe `JSValue` est utilisée par les moteurs JavaScript pour représenter des valeurs JavaScript. Cet objet est intéressant, car une `JSValue` peut être une référence vers un `JSObject` (`JSArrayBufferView`). Ces objets nous permettent donc de stocker des références de `JSObject` sur le tas de `FastMalloc`. Nous pouvons utiliser notre primitive de lecture relative pour retrouver ces références.

Cette technique, décrite ci-dessous, a été utilisée par Luca Todesco [8] afin de récupérer des références de `JSObject` sur le tas de `FastMalloc` :

1. Allouer plusieurs objets `JSArrayBufferView` ;

2. Stocker des références vers ces objets sur le tas de `FastMalloc` à l'aide de la méthode `Object.defineProperties`. À la fin de cette méthode, les deux tampons sont libérés, mais les références sont toujours présentes sur le tas. Il faut faire attention à ne pas allouer à nouveau ces deux tampons sous peine d'écraser les références ;
3. Parcourir le tas de `FastMalloc` et rechercher les références vers des `JSArrayBufferView` à l'aide de la primitive de lecture relative. La référence recherchée doit être allouée après l'objet `StringImpl` nous donnant une lecture relative, car il nous sera utile plus tard de pouvoir lire son contenu à partir de cette même primitive.

La figure 11 illustre la méthode utilisée pour obtenir l'adresse d'un objet `JSArrayBufferView`.

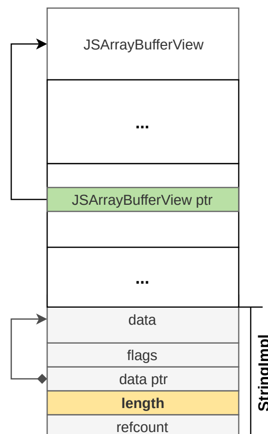

Fig. 11. Représentation mémoire de la recherche de référence vers un objet `JSArrayBufferView`.

4.5 La primitive de lecture/écriture relative

À partir de l'adresse d'un objet `JSArrayBufferView` ainsi que d'une primitive de décrément arbitraire, il est possible d'obtenir une primitive de lecture/écriture relative en suivant les étapes suivantes :

1. Déclencher à nouveau la vulnérabilité afin de réutiliser le décrément arbitraire ;
2. Utiliser le décrément arbitraire pour corrompre l'attribut `length` de l'objet `JSArrayBufferView` récupéré. L'objet corrompu peut

être retrouvé en vérifiant la taille de chaque `JSArrayBufferView` alloué. Celui qui a une taille excessive est l'objet corrompu.

La référence trouvée peut être utilisée pour lire et écrire au-delà des limites du tampon de donnée alloué par `FastMalloc`.

La figure 12 illustre la représentation mémoire obtenue après avoir corrompu un objet `JSArrayBufferView`.

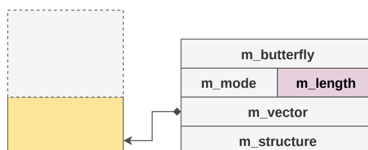

Fig. 12. Représentation mémoire d'un `JSArrayBufferView` corrompu.

4.6 La primitive de lecture/écriture arbitraire

L'objet `JSArrayBufferView` possède un attribut qui est une référence vers son tampon de donnée. Le but de cette partie va être de corrompre cet attribut afin d'obtenir une primitive de lecture/écriture arbitraire.

L'objet utilisé pour lire et écrire relativement en mémoire est alloué par *FastMalloc* alors que l'objet `JSArrayBufferView`, visé, est alloué par le *GC*. Cette différence nous empêche d'atteindre directement l'objet `JSArrayBufferView` à l'aide de nos primitives puisque rien ne nous garantie qu'il n'y a pas de pages mémoire non mappées entre ces deux tas. Cependant, nous connaissons l'adresse de l'objet `JSArrayBufferView` qui contient l'adresse du tampon utilisé pour lire et écrire relativement en mémoire. Etant donné que l'objet `JSArrayBufferView` est alloué après l'objet `StringImpl` (cf. 4.4), nous pouvons utiliser notre primitive de lecture relative pour récupérer l'adresse du tampon de données.

Il est maintenant possible d'atteindre l'un des `JSArrayBufferView` alloué à l'aide de la primitive de lecture et écriture relative et de corrompre le pointeur du tampon d'un autre `JSArrayBufferView`. Cela nous permet de lire et écrire des données arbitraires à une adresse arbitraire en utilisant le second `JSArrayBufferView`.

La figure 13 illustre la représentation mémoire permettant de lire et écrire à une adresse arbitraire à partir de deux objets `JSArrayBufferView`.

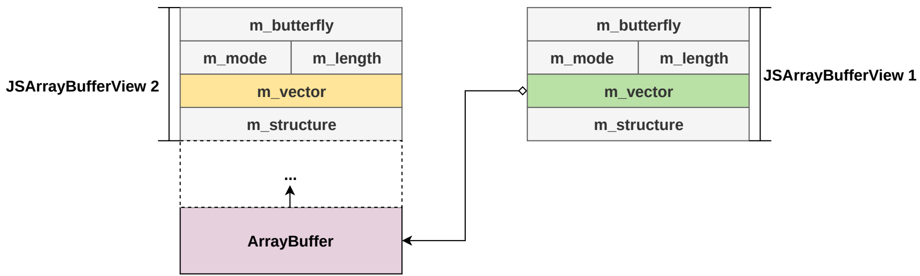

Fig. 13. Représentation mémoire de deux `JSArrayBufferView` corrompus.

4.7 Exécution de code

Dans le contexte du processus de rendu, la PlayStation 4 interdit de mapper des pages mémoires avec les permissions de lecture, écriture et exécution. Cependant, avec une primitive de lecture/écriture arbitraire, on peut contrôler le pointeur d'instructions. Par exemple, il est possible de corrompre l'un des pointeurs de fonction présent dans la table virtuelle de l'un de nos précédents `HTMLTextAreaElement` alloué. Lorsque la méthode JavaScript correspondante sera appelée, le processus effectuera un appel vers la valeur corrompue précédemment. À partir de là, il est possible d'utiliser des techniques de réutilisation de code telles que le *ROP* (*Return Oriented Programming*) ou le *JOP* (*Jump Oriented Programming*) afin d'implémenter le second maillon de la chaîne d'exploitation.

L'exploit complet est disponible sur le Github de Synacktiv [7].

4.8 Porter l'exploit sur la version 7.XX de la PlayStation 4

Notre exploit fonctionne correctement sur la version 6 de la PlayStation grâce à la faiblesse présente dans l'implémentation de l'*ASLR* qui nous a permis de prédire l'adresse d'objets *HTML*. L'adresse prédite est définie en dur dans l'exploit et a été identifiée grâce à l'exploit "bad-hoist". Cependant, sans connaissance préalable sur le mapping mémoire, la seule méthode pour déterminer l'adresse de l'un des `HTMLElement` alloués est de bruteforcer cette adresse.

Le bruteforce sur la PlayStation 4 est fastidieux étant donné que le navigateur a besoin d'une interaction utilisateur afin de redémarrer. Notre idée a été d'utiliser un Raspberry Pi pour émuler un clavier. Son but est d'entrer la touche *ENTER* à une fréquence régulière (5 secondes) afin de redémarrer le navigateur après qu'il ait planté. L'adresse bruteforcée est lue depuis un cookie mis à jour après chaque essai.

La figure 14 illustre notre tentative de bruteforce de l'*ASLR* de la PlayStation 4.

Fig. 14. Tentative de bruteforce de l'*ASLR*.

Malheureusement, cette méthode n'a pas donné de résultats. Lors de l'écriture de l'exploit, nous n'avions aucune connaissance sur le mapping mémoire du tas sur la version 7 de la PS4. Nous avons tenté de réduire le nombre de possibilités en partant du principe que le mapping mémoire n'avait pas changé entre les versions 6 et 7. Cette piste n'a pas donné de résultats.

5 Conclusion

Environ une semaine après avoir publié notre exploit en décembre 2020, un dénommé sleirsgoevy publie sur Github [6] une version modifiée de notre exploit fonctionnant sur les versions 7 de la PlayStation 4. Après étude de ce dernier, il semblerait que seulement deux modifications aient été faites :

1. L'adresse codée en dur a été modifiée afin de permettre le contournement de l'*ASLR* sur les versions 7 de la PS4. Celle-ci a été obtenue par bruteforce en s'inspirant de la technique décrite en section 4.8 ;
2. La recherche des `JSValue` à l'aide de la primitive de lecture relative ne fonctionnait pas sur la version 7 de la PS4. Ce problème semblait

être lié au fait que le tas ne devait pas avoir la même disposition mémoire que celle obtenue lors du développement de l'exploit sur la version 6. Pour corriger ce problème, l'auteur a alloué plus d'objets afin de s'assurer que l'objet `JSArrayBufferView` recherché ait bien été alloué après l'objet `StringImpl` utilisé. Il s'agit d'un ajout fonctionnel qui n'améliore pas la stabilité de l'exploit d'origine.

Le reste de l'exploit est identique à notre version.

Grâce à nos efforts communs, nous avons écrit le premier exploit navigateur pour les versions 7 de la PlayStation 4, contribuant ainsi au premier jailbreak public sur la version 7. En effet, peut après la publication de l'exploit et de son portage sur la version 7, il a été combiné avec un exploit kernel.

Selon des informations publiées sur Internet, la PlayStation 5 disposerait d'un navigateur mais celui-ci n'est pas accessible depuis le menu des applications [13]. Cela implique que l'idée d'une chaîne reposant sur une vulnérabilité dans le navigateur pour obtenir de l'exécution de code sur la console serait toujours envisageable.

Références

1. bad_hoist. `https://github.com/Fire30/bad_hoist`.
2. Breaking down qwerty's ps4 4.0x webkit exploit. `https://github.com/Cryptogenic/Exploit-Writeups/blob/master/PS4/4.0x%20WebKit%20Exploit%20Writeup.md`.
3. Protect frames during style and layout changes. `https://github.com/WebKit/WebKit/commit/a7163fe343a407f4712b90e9b0186db237361f65`.
4. Ps4 4.55 / freebsd bpf kernel exploit writeup. `https://github.com/Cryptogenic/Exploit-Writeups/blob/master/FreeBSD/PS4%204.55%20BPF%20Race%20Condition%20Kernel%20Exploit%20Writeup.md`.
5. Ps4 5.05 / freebsd bpf 2nd kernel exploit writeup. `https://github.com/Cryptogenic/Exploit-Writeups/blob/master/FreeBSD/PS4%205.05%20BPF%20Double%20Free%20Kernel%20Exploit%20Writeup.md`.
6. Ps4 jailbreak. `https://github.com/sleirsgoevy/ps4jb`.
7. Repertoire github de l'exploit ps4 sur les versions 6. `https://github.com/synacktiv/PS4-webkit-exploit-6.XX`.
8. setattributenodens use-after-free webkit exploit. `https://github.com/Cryptogenic/Exploit-Writeups/blob/master/WebKit/setAttributeNodeNS%20UAF%20Write-up.md`.
9. Webkit sources. `https://doc.dl.playstation.net/doc/ps4-oss/webkit.html`.
10. Eloi Benoist-Vanderbeken and Fabien Perigaud. Wen eta jb ? a 2 million dollars problem. `https://www.sstic.org/media/SSTIC2019/SSTIC-actes/WEN_ETA_JB/SSTIC2019-Article-WEN_ETA_JB-benoist-vanderbeken_perigaud.pdf`.

11. Sam Brown. Some brief notes on webkit heap hardening. https://labs.f-secure.com/archive/some-brief-notes-on-webkit-heap-hardening/.
12. Andy Nguyen. Cve-2020-7457. https://hackerone.com/reports/826026.
13. Kyle Orland. The playstation 5 has a hidden web browser ; here's how to find it. https://arstechnica.com/gaming/2020/11/the-playstation-5-has-a-hidden-web-browser-heres-how-to-find-it/.
14. Wang Yong. Thinking outside the jit compiler : Understanding and bypassing structureid randomization with generic and old-school methods. https://i.blackhat.com/eu-19/Thursday/eu-19-Wang-Thinking-Outside-The-JIT-Compiler-Understanding-And-Bypassing-StructureID-Randomization-With-Generic-And-Old-School-Methods.pdf.