FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Přetečení bufferu na haldě
Student:	Michal Bambuch
Vedoucí:	Ing. Josef Kokeš
Studijní program:	Informatika
Studijní obor:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Seznamte se s problematikou přetečení bufferu, jeho příčinami a následky. Porovnejte přetečení na zásobníku a na haldě.
Proveďte rešerši známých útoků na haldu knihovny glibc.
Vyhodnoťte, které verze knihovny glibc jsou zranitelné kterými útoky.
Připravte vhodné virtuální prostředí a demonstrujte v něm vybrané útoky.
Pro zvolené už nefunkční útoky nastudujte, jakým způsobem jim paměťový manažer brání. Diskutujte účinnost těchto opatření.

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Pavel Tvrdík, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 25. ledna 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Přetečení bufferu na haldě

Michal Bambuch

Katedra počítačových systémů
Vedoucí práce: Ing. Josef Kokeš

13. května 2019

Poděkování

Na tomto místě bych rád poděkoval vedoucímu mé práce Ing. Josefu Kokešovi za vedení a pomoc při jejím vypracování. Dále bych rád poděkoval mé rodině za veškerou podporu při studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 13. května 2019 ……

Odkaz na tuto práci

Bambuch, Michal. *Přetečení bufferu na haldě*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato bakalářská práce se zabývá přetečením bufferu na haldě a známými útoky na haldu knihovny glibc. Cílem práce je vytvořit virtuální prostředí a implementovat v něm vybrané útoky. Jako virtuální prostředí byla zvolena instalace Ubuntu 18.04 LTS v nástroji VirtualBox. Pro demonstraci byly zvoleny 4 útoky – unlink makro, House of Force a House of Spirit z Malloc Maleficarum a House of Einherjar. Poslední z nich je proveditelný na 2 roky staré verzi glibc 2.25, House of Force a House of Spirit jsou proveditelné na glibc 2.27 instalované přímo ve virtuálním prostředí.

Klíčová slova přetečení bufferu, přetečení na haldě, Linux, knihovna glibc, správa paměti, unlink makro, Malloc Maleficarum

Abstract

This bachelor's thesis addresses heap buffer overflow and known exploits of the glibc library's heap. The aim of the thesis is to create a virtual environment and implement selected exploits. The set environment is Ubuntu 18.04 LTS virtualised in VirtualBox. The following 4 exploits have been selected for demonstration: the unlink macro, the House of Force and the House of Spirit from the Malloc Maleficarum and finally the House of Einherjar. The last one is reproducible in a two-year-old glibc 2.25. The two Malloc Maleficarum's exploits are reproducible in glibc 2.27 installed directly on the virtual environment.

Keywords buffer overflow, heap overflow, Linux, glibc library, memory management, unlink macro, Malloc Maleficarum

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Úvod

Téma přetečení bufferu je ve světě počítačové bezpečnosti stále aktuální. Užívání programovacích jazyků přímo pracujících s pamětí přirozeně vede k možným chybám. Špatná práce s pamětí může mít až fatální následky, spuštění cizího kódu a ovládnutí celého systému není výjimečné.

Tato práce má seznámit zejména programátory v jazyce C s problematikou přetečení bufferu. To může být dvojího druhu, odehrává se buď na zásobníku, nebo na haldě. Hlavní část práce se zaměřuje na problematiku haldy zejména ve spojitosti s knihovnou glibc, která je použita jako základní knihovna jazyka C ve většině operačních systémů GNU/Linux.

Cílem této bakalářské práce je provedení rešerše známých útoků na haldu knihovny glibc a vyhodnocení, které verze knihovny glibc jsou vůči těmto útokům zranitelné. Dalším cílem je vytvoření virtuálního prostředí a v něm demonstrace vybraných útoků.

První kapitola je věnována obecnému seznámení s problematikou přetečení, s jeho možnými příčinami a následky. Druhá kapitola je zaměřena na organizaci haldy knihovny glibc a popisuje její základní prvky, jejichž znalost je nutná pro pochopení fungování paměťového manažeru. Třetí kapitola je rešerší známých útoků a ve čtvrté kapitole je popsáno praktické provedení vybraných útoků ve vytvořeném virtuálním prostředí.

Kapitola 1

Přetečení bufferu

Přetečením bufferu se označuje situace, kdy je do bufferu o určité velikosti zapsáno více dat, než je jeho kapacita [1]. V programovacích jazycích jako je Java nebo C# tento problém běžně nenastane, protože jsou automaticky kontrolovány velikosti polí. Ovšem v jazycích používajících přímý přístup do paměti (například C/C++) si programátor na tuto chybu musí dávat pozor sám.

Přetečení bufferu bylo nejčastějším typem bezpečnostní zranitelnosti v devadesátých letech [2]. Tento typ zranitelnosti zneužil již v roce 1988 jeden z prvních počítačových virů šířící se přes internet – červ Morris [3], který (mimo jiné) pro své šíření využíval přetečení bufferu u programu finger. Autorem viru byl 21letý student Cornellovy univerzity Robert Tappan Morris. Vir napadl zhruba 6 000 počítačů. Způsoboval ztrátu výkonu napadeného počítače, ale nijak neupravoval ani nemazal uživatelská či systémová data.

Detailní popis problematiky přetečení bufferu přinesl článek *„Smashing The Stack For Fun And Profit“* [1] publikovaný v roce 1996. Ukázal a popsal i praktické ukázky napadení zranitelného programu.

V dnešní době je zneužití tohoto typu zranitelnosti náročnější než dříve kvůli pokročilejším způsobům ochrany paměti. Avšak pohledem do databáze zranitelností CVE lze zjistit, že i v dnešní době je tato zranitelnost velice častá.

1.1 Možné příčiny přetečení bufferu

K přetečení bufferu může dojít z mnoha různých důvodů. Při kopírování dat mezi dvěma buffery je třeba dbát na to, aby cílový buffer byl dostatečně velký. Při načítání vstupu od uživatele je třeba nastavit maximální velikost přijatých dat, aby uživatel nemohl zadat víc dat, než bylo předpokládáno. Při použití knihovních funkcí je třeba dobře znát jejich funkčnost, aby se nechovaly jinak, než bylo zamýšleno. Některé funkce je dobré raději nepoužívat. Celkově je

třeba být při přímé práci s pamětí na pozoru. V dalších podkapitolách budou ukázány časté[1] chyby, které mohou vést k přetečení bufferu při programování v jazyce C.

1.1.1 Špatné kopírování řetězců – funkce strcpy()

Kopírování řetězců z jednoho bufferu do druhého je velmi běžnou operací. Při neověřování velikosti cílového bufferu se může stát, že se bude kopírovat z většího bufferu do menšího. V takovém případě může dojít k přetečení bufferu.

Pro kopírování řetězců nabízí základní knihovna jazyka C funkci `strcpy()` s následujícím prototypem:

```
char *strcpy(char *dest, const char *src);
```

Výpis kódu 1.1: Prototyp funkce `strcpy()`

Tato funkce má dva parametry – ukazatel na zdrojový řetězec `*src` a ukazatel na cílový řetězec `*dest`. Funkce nijak nehlídá, jestli není cílový buffer menší než zdrojový. To je zodpovědností programátora. Její špatné použití může vypadat následovně:

```
char srcBuff[16] = "Buffer_overflow";
char destBuff[10];
strcpy(destBuff, srcBuff);
```

Výpis kódu 1.2: Ukázka nebezpečného použití funkce `strcpy()`

Před provedením kopírování nebyla ověřena dostatečná velikost cílového bufferu. Jelikož je cílový buffer menší než zdrojový, po provedení kopírování přeteče cílový buffer `srcBuff[]` o 6 znaků. Zdánlivým řešením by mohlo být použití „bezpečnější“ funkce `strncpy()`, která jako další argument přijímá maximální počet bytů ke kopírování.

```
char srcBuff[16] = "Buffer_overflow";
char destBuff[10];
strncpy(destBuff, srcBuff, 10);
```

Výpis kódu 1.3: Ukázka nebezpečného použití funkce `strncpy()`

Nyní sice nedošlo k přetečení bufferu, ale došlo k vytvoření nové bezpečnostní chyby. Řetězce znaků jsou v jazyce C standardně ukončené nulovým bytem (`'\0'`). V manuálu funkce `strncpy()` se píše, že pokud není v prvních n bytech řetězce nulový byte, zkopírovaný řetězec nebude ukončen nulou [4].

[1] Tento seznam rozhodně není vyčerpávající, ukazuje jen běžné omyly.

A to se stalo v tomto případě. Následné použití knihovních funkcí na tento řetězec vede k nedefinovanému chování, které je možné prakticky zneužít.

Možným řešením je použití alternativních funkcí, které však nebývají součástí standardní knihovny. S tím bohužel souvisí omezení portability programu. Jednou z možností je funkce `strlcpy()`, která pochází z projektu OpenBSD. Tato funkce oproti `strncpy()` vždy zajišťuje správné zakončení řetězce nulovým bytem. Je dostupná v mnoha Unixových systémech, avšak nikdy nebyla přidána do knihovny glibc. V systémech GNU/Linux ji lze využívat použitím knihovny libbsd. Na platformě Microsoft Windows lze využít knihovní funkci `strcpy_s()`.

1.1.2 Spojování řetězců – funkce strcat()

Ke spojování řetězců poskytuje standardní knihovna funkci `strcat()`. Podobně jako u funkce `strcpy()` je třeba dávat pozor na dostatečnou velikost cílového bufferu. Prototyp funkce je následující:

```
char *strcat(char *dest, const char *src);
```

Výpis kódu 1.4: Prototyp funkce `strcat()`

Tato funkce na konec řetězce `dest[]` připojí řetězec `src[]`. Opět neověřuje, zdali je cílový řetězec dostatečně velký.

Kromě funkce `strcat()` nabízí standardní knihovna funkci `strncat()`. Ta jako další parametr požaduje počet znaků, které se mají zkopírovat. Na rozdíl od `strncpy()` v případě delšího řetězce finální řetězec správně ukončí nulovým bytem, ale dodá ho až jako $(n+1)$. znak. V případě, že programátor nezná toto chování, může dojít k přetečení o jeden byte (*přetečení off-by-one*) – více v kapitole 1.1.7.

Bezpečnější alternativou pro Unixové operační systémy může být funkce `strlcat()`, pro platformu Microsoft Windows funkce `strcat_s()`.

1.1.3 Načítání dat – funkce gets()

Pro načítání vstupu od uživatele nabízí základní knihovna hned několik funkcí. Jednou z nich je i funkce `gets()` s následujícím prototypem:

```
char *gets(char *s);
```

Výpis kódu 1.5: Prototyp funkce `gets()`

Funkce zapíše do bufferu `s[]` řádek ze standardního vstupu a ukončí ho nulovým bytem. Problémem je, že nelze nijak omezit velikost vstupního řetězce a nedá se tedy jakýmkoliv způsobem zabránit potenciálnímu přetečení bufferu.

Tato funkce je tedy vždy nebezpečná a neměla by se používat – dle manuálu: *„Never use this function“* [5]. Jedná se o jedinou funkci základní knihovny,

kterou nelze použít bezpečně. Jako její náhradu lze použít funkci `fgets()`, která dle jednoho z parametrů omezuje maximální počet načtených znaků. Tento parametr je důležité správně zvolit, aby nedošlo například k přetečení o jeden byte.

1.1.4 Načítání dat – funkce scanf()

Rodina funkcí `scanf()` slouží k načítání vstupu a jeho případné konverzi dle zadaného formátovacího řetězce. Patří do ní následující funkce:

```
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);

int vscanf(const char *format, va_list ap);
int vsscanf(const char *str, const char *format, va_list ap);
int vfscanf(FILE *stream, const char *format, va_list ap);
```

Výpis kódu 1.6: Prototypy funkcí z rodiny `scanf()`

Funkce `scanf()` slouží k načítání ze standardního vstupu, funkce `fscanf()` k načítání ze souboru a funkce `sscanf()` k načítání z textového řetězce. Funkce s předponou `v-` jsou jejich obdoby, pouze cíl zápisu se předává pomocí struktury `va_list`.

Opět je třeba dávat pozor, aby byl cílový buffer dostatečně velký. Typickým příkladem špatného použití funkce `scanf()` může být načítání řetězce ze základního vstupu bez omezení jeho maximální délky.

```
char buffer[20];
scanf("%s", buffer);
```

Výpis kódu 1.7: Nebezpečné použití funkce `scanf()`

V tomto případě, pokud uživatel zadá řetězec delší než 19 znaků, dojde k přetečení bufferu. Obrana je jednoduchá, stačí omezit maximální počet znaků, které může formátovací znak načíst, na 19 (20. je nulový byte).

```
char buffer[20];
scanf("%19s", buffer);
```

Výpis kódu 1.8: Správné použití funkce `scanf()`

Ve standardní knihovně neexistují „bezpečnější“ alternativy k těmto funkcím. Na platformě Microsoft Windows lze využít funkce s příponou `-_s` jako `scanf_s()`, `fscanf_s()` apod., u kterých je povinnost zadat velikost cílového bufferu.

1.1.5 Vypsání dat – funkce sprintf()

Rodina funkcí `printf()` slouží k vypsání dat na výstup dle zadaného formátovacího řetězce. Je opakem rodiny `scanf()` sloužící k načtení vstupu. Speciálním případem je funkce `sprintf()` s následujícím prototypem:

```
int sprintf(char *str, const char *format, ...);
```

Výpis kódu 1.9: Prototyp funkce `sprintf()`

Tato funkce zapíše do řetězce `str[]` výstup dle zadaného formátovacího řetězce. Velikost cílového řetězce není nijak ověřována. Programátor musí zajistit, že nebude zapsán příliš velký řetězec. Lepší alternativou je funkce `snprintf()`, která navíc jako 2. argument vyžaduje velikost bufferu, do kterého bude zapisovat. Obdobně místo funkce `vsprintf()` (místo proměnného počtu argumentů se předává ukazatel na strukturu argumentů `va_list`) je lepší použít funkci `vsnprintf()`.

Špatným použitím funkcí sloužících k vypsání formátovaného vstupu lze způsobit další bezpečnostní chybu – chybu formátovacích řetězců (*format string vulnerability*). Ta může vést k odhalení informací o procesu (vypsání obsahu paměti), ke způsobení pádu programu (odepření služby – *denial of service attack*) nebo k zapsání na takřka libovolné místo v paměti [6]. Vypisovacím funkcím nesmí být nikdy předán pouze jeden parametr obsahující proměnnou k vypsání jako lze vidět na následujícím příkladu:

```
char userInput[30];
scanf("%29s", userInput);
printf(userInput);          //zde chyba!
```

Výpis kódu 1.10: Ukázka zranitelnosti formátovacích řetězců

Ve výše uvedeném příkladě uživatel ovlivňuje řetězec, který je předaný funkci `printf()` k vypsání. Pokud zadá běžný text, je tento text korektně vypsán. Ale pokud použije nějaký formátovací řetězec, je tento řetězec použit a vykonán.

Funkce z rodiny `printf()` určí počet předaných parametrů dle formátovacího řetězce, respektive dle jeho řídících znaků. V případě, že počet parametrů je menší než počet formátovacích znaků, pracuje funkce s cizí pamětí. Pokud potencionální útočník použije například několik formátovacích řetězců `%p`, dojde k vypsání části paměti zásobníku. Tím může útočník získat důvěrné informace. Ještě problematičtější je formátovací řetězec `%n`, který zapíše do dané proměnné počet dosud vypsaných znaků.

Pro reálný útok se využívá dalších možností, které formátovací řetězce nabízí. Například POSIX standard zavádí možnost změny pořadí čtení argumentů – 50. argument lze přečíst pomocí formátovacího řetězce `%50$p` a není třeba použít jako řetězec 50x `%p` [6, 7].

Řešením této chyby je správné použití funkce `printf()`. Funkci nesmí být nikdy předán pouze jeden parametr, vypsání řetězce je nutné provést následovně:

```
printf("%s", userInput);
```

Výpis kódu 1.11: Bezpečné použití funkce `printf()`

1.1.6 Další rizikové funkce základní knihovny

Vyjma výše uvedených funkcí existují i další rizikové funkce. Jedná se zejména o funkce, které přímo pracují s pamětí (např. `memcpy()`, `bcopy()`, `memccpy()`, `memmove()`) a které mají na starosti načítání dat (např. `fgets()`, `read()`) [8].

Pro zabránění přetečení bufferu je potřeba vždy ověřovat velikosti kopírovaných dat a omezovat maximální velikost načítaných dat. Je třeba znát do detailu chování používaných funkcí. Celkově musí být programátor opatrný.

1.1.7 Přetečení o jeden byte (Off-by-one overflow)

Přetečení o pouhý jeden byte je programátorskou chybou, která nastává při špatném vyhodnocení mezní situace. I takové malé přetečení může představovat vážnou bezpečnostní chybu [9]. Tato chyba může být například způsobená špatnou podmínkou v cyklu:

```
for(i=0; i<=x; i++){}  //off-by-one overflow
for(i=0; i<x; i++){}   //správně
```

Výpis kódu 1.12: Ukázka přetečení o jeden byte v cyklu

Obdobnou chybou může být i špatný přístup do pole. Ukázka představuje špatný přístup k poslednímu prvku:

```
arr[sizeof(arr)];     //off-by-one overflow
arr[sizeof(arr)-1];   //správně
```

Výpis kódu 1.13: Ukázka přetečení o jeden byte v přístupu do pole

Další možností, kdy může chyba nastat, je práce s řetězci. Programátor nesmí zapomenout na nulový byte, který některé funkce přidávají za zpracovaný řetězec. Například při načítání řetězce do pole délky 20 znaků je potřeba nastavit maximální počet načtených znaků na 19.

```
char str[20];
scanf("%20s", str);   //off-by-one overflow
scanf("%19s", str);   //správně
```

Výpis kódu 1.14: Ukázka přetečení o jeden byte u načítání vstupu

1.2 Přetečení bufferu na zásobníku

Ukázkové přetečení bufferu na zásobníku může vypadat následovně. Nechť je na zásobníku pole pro řetězec o délce 12 znaků. Následně je do něj zapsáno znaků 16:

```
char str[12];
strcpy(str, "buffer overflow");
```

Výpis kódu 1.15: Ukázka přetečení bufferu na zásobníku

Po provedení kopírovaní byly zapsány 4 byty za řetězec `str[]`. Mohlo dojít k přepsání nějaké jiné uživatelské proměnné, ale mohlo také dojít k přepsaní proměnné ovlivňující vykonání programu.

1.2.1 Struktura zásobníku

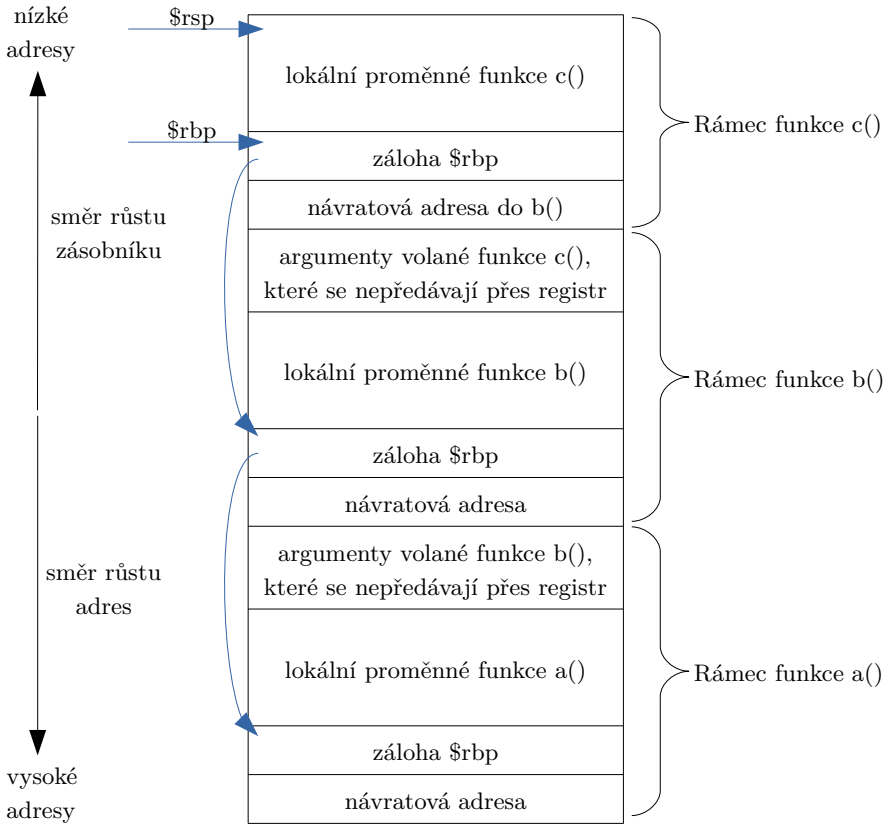
Zásobník je datová struktura obsahující jednak lokální proměnné definované uvnitř funkcí, ale i pomocná data potřebná k volání funkcí, jako jsou jejich parametry, návratová adresa či zálohy různých registrů. Tato struktura funguje na principu LIFO (*last in, first out*), neboli poslední prvek, který byl do zásobníku přidán, je vyjmut jako první. Pro operace se zásobníkem existují 2 základní instrukce, `push` slouží k přidání proměnné na zásobník a `pop` k vyjmutí. Na vrchol zásobníku ukazuje registr `$rsp` (stack pointer). Takto je registr nazván na 64bitové architektuře x86_64, na 32bitové architektuře x86 se jmenuje `$esp`. Jelikož zásobník „roste“ od větších adres k menším, při operaci `push` je hodnota ukazatele v registru `$rsp` zmenšena, při operaci `pop` naopak zvětšena.

Pro ukázku se předpokládá 64bitová architektura x86_64 bez ochran zásobníku (jako je například StackGuard). Obrázek 1.1 zobrazuje zásobník programu, kde funkce `a()` zavolala funkci `b()` a ta zavolala funkci `c()`.

Na ukázce zásobníku se nachází tři rámce – každý představuje jednu funkci. Aktuální rámec funkce `c()` se nachází na vrcholu zásobníku. Na spodek rámce ukazuje registr `$rbp` (base pointer). Při volání funkce `c()` funkcí `b()` nazýváme funkci `b()` jako volající a funkci `c()` jako volanou. Existuje několik volacích konvencí, které určují, jak má volání funkcí probíhat. V Linuxových operačních systémech se využívá konvence *System V AMD64 ABI* [10].

Dle této konvence má volající za úkol zařídit předání parametrů volanému. Parametry jsou dle typu a počtu předány pomocí registrů nebo přes zásobník. Poté je na zásobník uložena návratová adresa a `$rip` (instruction pointer – čítač instrukcí) je změněn na volanou funkci. Tím je funkce zavolána.

Volaný nejdříve provede zálohu registrů, které dle konvence nesmí měnit. Dojde tedy minimálně k uložení zálohy registru `$rbp`. Poté je do `$rbp` uložen ukazatel na tuto zálohu. Registr je následně používán k adresaci lokálních pro-



Obrázek 1.1: Rámce funkcí na zásobníku

měnných na zásobníku. Následně jsou na zásobník uloženy parametry předané pomocí registrů.

Po dokončení volané funkce volaná funkce uloží návratovou hodnotu do patřičného registru (nebo místa v paměti přiděleným volajícím), uklidí zásobník a na závěr obnoví registr `$rbp` ze zálohy a změní `$rip` podle uložené návratové adresy. Tím je zásobník v původním stavu před voláním a volající může pokračovat.

1.2.2 Následky přetečení

V případě, že dojde k přetečení bufferu, může útočník změnit paměť za buffrem – ten roste směrem k vyšším adresám. Může přepsat obsah lokálních proměnných a tím ovlivnit logiku programu. Dále může přepsat zálohu registru `$ebp`. Největšího úspěchu ovšem dosáhne přepsáním návratové adresy.

Přepsání návratové adresy na náhodnou hodnotu většinou vede k pádu programu, protože daná stránka paměti pravděpodobně nebude vůbec existovat či bude obsahovat nesmyslné instrukce. Nejedná se o zanedbatelnou věc – útočník tím může způsobit útok odepření služby (*denial of service attack*).

Důmyslnější úpravou může útočník spustit vlastní kód. Někam do paměti programu nahraje svůj strojový kód (instrukce pro procesor), což může dokázat validním zápisem do nějakého jiného bufferu nebo například do bufferu trpícího přetečením. Vhodným přepsáním návratové adresy pak může tento vlastní kód nechat vykonat.

Kód může být sestaven například z jednoduchého systémového volání – instrukce syscall 11. Ta představuje volání fukce `execve()` sloužící k spuštění jiného programu. Zavoláním této funkce s parametrem `"/bin/sh"` získá útočník na počítači konzoli shellu. V případě, že daný program běžel s právy superuživatele, získal i plný přístup k systému.

1.2.3 Metody ochrany paměti

V dnešní době se používá mnoho různých metod ochran paměti. Ty komplikují možnosti úspěšného útoku, avšak ne vždy jej znemožňují. Mezi tyto metody obrany se řadí kanárci, NX-bit (DEP) a ASLR.

Kanárci Název vznikl podle obdobného způsobu používání kanárků k obraně před otravou v dolech. Kanárek (náhodná hodnota) bývá umístěn typicky mezi lokální proměnné a návratovou adresu. Pokud je kanárek změněn, je program před návratem z funkce ukončen. Tento způsob ochrany mají implementován běžné kompilátory jako je GCC, LLVM či Microsoft Visual Studio.

NX-bit (Non eXecutable bit) Tato technologie implementovaná u moderních procesorů slouží k tzv. značkování jednotlivých stránek paměti, kde je nastavené, zdali může být z dané stránky spuštěn kód. Funkčnost této obrany je závislá na podpoře operačního systému. V systémech Microsoft Windows je tato funkce označovaná jako DEP (data execution prevention).

ASLR (Address space layout randomization) Tato technika má zatajovat útočníkovi umístění zásobníku, haldy, kódu i knihoven, které jsou do paměti nahrány s určitým posunutím, každý běh programu jsou umístěny na jiné adrese. Tuto obranu má na starosti operační systém.

1.3 Přetečení bufferu na haldě

K přetečení bufferu na haldě může dojít z obdobných příčin jako u přetečení na zásobníku. Jediný rozdíl oproti zásobníku je v tom, že se buffer nachází v jiné části paměti – na haldě.

Halda slouží pro uložení větších polí, bufferů a jiných datových struktur. Místo na haldě je dynamicky přidělované, programátor si musí o daný blok paměti zažádat pomocí funkce `malloc()`, která mu vrátí ukazatel do alokované

paměti (pokud byla alokace úspěšná). Jeho povinností je následnou paměť uvolnit poté, co ji nebude již nadále potřebovat, pomocí funkce `free()`.

Halda – neboli struktura, ve které jsou uložené tyto alokované bloky, má mnohem komplikovanější strukturu než zásobník. Tato struktura bude podrobně popsána v následující kapitole. Na haldě nejsou uložené zálohy registrů ani návratové hodnoty, ale vhodným přepsáním interních struktur haldy, které jsou umístěny mezi jednotlivými alokovanými bloky, lze paměťový manažer přimět k nevalidnímu chování.

Často je k těmto útokům potřeba ještě další programátorova chyba, jako je například volání funkce `free()` na již uvolněné bloky (chyba *double free*). Jejich výsledkem může být nesprávné vrácení ukazatele funkcí `malloc()` či dokonce zápis do libovolného místa v paměti.

Možné způsoby exploitace haldy (používané knihovnou glibc) a obrany proti nim budou popsány v dalších kapitolách.

1.4 Srovnání přetečení na haldě a na zásobníku

Zatímco zásobník mívá stejnou strukturu – nachází se v něm uživatelská data společně se zálohami registrů, argumenty funkcí, návratovými adresami, struktura haldy závisí na dané implementaci (i verzi) základní knihovny jazyka C resp. paměťového manažeru.

Při exploitaci zásobníku útočník využívá toho, že jsou společně uložena lokální data s návratovými adresami, zálohami registrů a argumenty funkcí. Struktura zásobníku bývá univerzální, možný útok tedy také.

Exploitace haldy bývá složitější, útočník musí znát (mnohem komplexnější) fungování a strukturu konkrétního paměťového manažeru. Při exploitaci útočník využívá smíchání uživatelských dat s vnitřními daty paměťového manažeru. Avšak paměťové manažery se vyvíjejí a opravují možné způsoby exploitace. Konkrétní způsob útoku může být funkční pouze na určité verzi.

Existují jednoduché ochrany zásobníku (výše zmínění kanárci), ty však ale nejsou použitelné pro ochranu haldy. Ta kvůli své složitosti vyžaduje komplexnější přístup.

Paměťový manažer (resp. programovací jazyk C) je navržen pro vysokou efektivnost, která vyžaduje přímou práci s pamětí. Každá bezpečnostní kontrola dat tuto efektivnost snižuje. Způsoby exploitace haldy vždy vyžadují programátorskou chybu – umožnění přetečení bufferu, a tím nevalidní přepsání dat paměťového manažeru. Je otázkou k diskusi, do jaké míry by měl paměťový manažer bránit tomuto zneužití za cenu efektivnosti programovacího jazyka.

Kapitola 2

Organizace haldy glibc

Knihovna glibc (GNU C Library) bývá používána jako základní systémová knihovna pro systémy GNU, GNU/Linux a další systémy využívající linuxový kernel [11]. Kromě implementace standardní knihovny jazyka C poskytuje rozhraní pro POSIX. Její první verze glibc 0.1 vyšla 8. října 1991. Vývoj knihovny je aktivní, nová verze vychází každého půl roku. Nejnovější verze 2.29 vyšla 31. ledna 2019. V současnosti je glibc používána jako základní systémová knihovna v nejpoužívanějších distribucích GNU/Linux jako je Arch Linux, CentOS, Debian, Fedora, Linux Mint, OpenSUSE či Ubuntu.

Pro práci s pamětí poskytuje standardní knihovna jazyka C tyto funkce – `malloc()`, `free()`, `calloc()` a `realloc()`. Implementace správy paměti je v knihovně glibc založena na alokátoru od Douga Leaho, který bývá označován jako dlmalloc[2], a jeho následné úpravě pro vícevláknové aplikace od Wolframa Glogera pojmenované ptmalloc2[3].

Paměť získaná těmito funkcemi je organizovaná pomocí složitějších datových struktur. Paměť jednoho procesu je organizovaná pomocí arén. Aréna se může skládat z několika hald a halda je složená ze zřetězených chunků [12]. V následujících podkapitolách budou tyto struktury podrobně popsány.

2.1 Arény

Při zobecněném pohledu na správu paměti si lze arény představit jako nejvíce nadřazené objekty. Arény obstarávají správu paměti pro běžící program. V minulosti se používala pouze jedna aréna pro celý program, dnes se pro efektivní správu paměti vícevláknových aplikací používá arén více. Neplatí ovšem, že by každé vlákno mělo svoji vlastní arénu. Počet arén je omezen dle počtu jader procesoru [13].

[2] `http://g.oswego.edu/dl/html/malloc.html`
[3] `http://www.malloc.de/en/`

Existují 2 druhy arén – aréna hlavní a arény ostatní. Hlavní aréna odpovídá prvotní haldě programu a skládá se pouze z této jedné haldy. Ukazuje na ní statická struktura `main_arena`[4]. Ostatní arény se skládají z více hald, které jsou alokovány pomocí funkce `mmap()`. Každá aréna je definovaná strukturou `malloc_state` (hlavička arény):

```
struct malloc_state{
  __libc_lock_define (, mutex);
  int flags;
  int have_fastchunks;

  mfastbinptr fastbinsY[NFASTBINS];
  mchunkptr top;
  mchunkptr last_remainder;
  mchunkptr bins[NBINS * 2 - 2];
  unsigned int binmap[BINMAPSIZE];
  struct malloc_state *next;
  struct malloc_state *next_free;

  INTERNAL_SIZE_T attached_threads;
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};
```

Výpis kódu 2.1: Hlavička arény – struktura `malloc_state` (zdrojový soubor glibc v. 2.29 /malloc/malloc.c, ř. 1665-1707 kráceno)

Struktura obsahuje mutex pro serializaci přístupů k aréně, příznaky (flagy) popisující vlastnosti arény, ukazatel na top chunk a last remainder chunk, ukazatele na koše (bins) s chunky. Dále obsahuje ukazatel `*next` na další arénu a ukazatel `*next_free` na další volnou arénu. Jednotlivé arény jsou zřetězené ve spojovém seznamu, každá aréna má ukazatel na další v pořadí. Nakonec obsahuje informace o počtu přiřazených vláken a využití paměti.

2.2 Haldy

Halda představuje jednu souvislou část paměti, která se dále dělí na chunky, které je možno alokovat. Každá halda náleží právě do jedné arény. S výjimkou hlavní arény se může aréna skládat z více hald. V momentě, kdy je jedna halda arény zaplněna, je vytvořena pomocí funkce `mmap()` halda druhá, na které se alokuje další paměť.

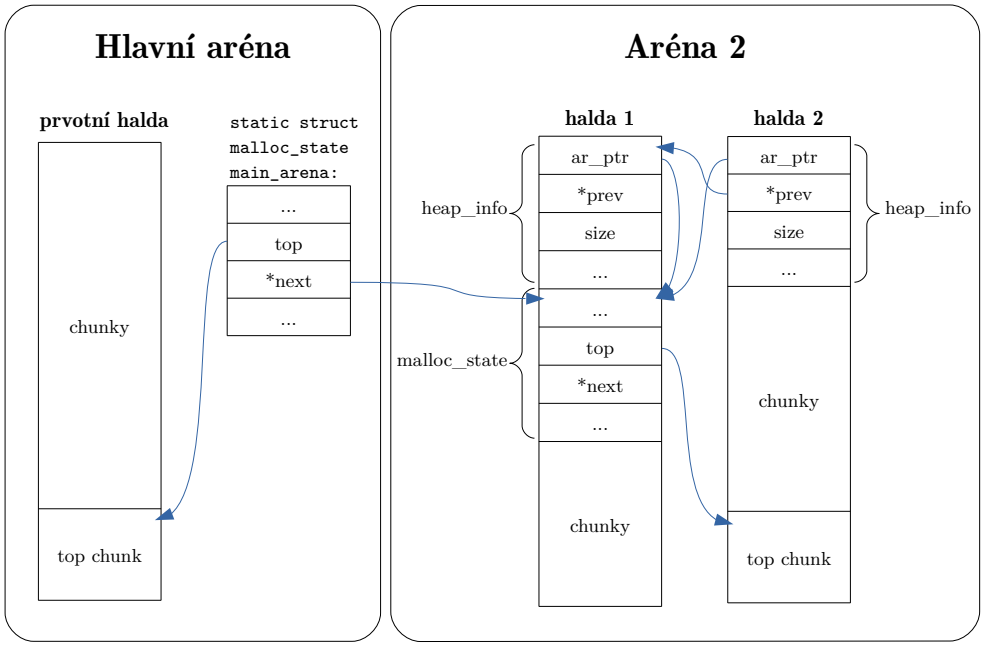
[4]zdrojový soubor glibc 2.28 /malloc/malloc.c, řádek 1752

Informace o haldě drží struktura `heap_info` (hlavička haldy). Tuto strukturu má každá halda vyjma prvotní haldy v hlavní aréně. Struktura je definovaná následovně:

```
typedef struct _heap_info {
  mstate ar_ptr;
  struct _heap_info *prev;
  size_t size;
  size_t mprotect_size;
  char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK];
} heap_info;
```

Výpis kódu 2.2: Hlavička haldy – struktura `heap_info` (zdrojový soubor glibc v. 2.29 /malloc/arena.c, ř. 53-64 kráceno)

Ukazatel `ar_ptr` ukazuje na arénu, do které daná halda patří, `*prev` ukazuje na předchozí haldu. Dále struktura obsahuje informace o velikosti haldy a výplň pro správné zarovnání v paměti.



Obrázek 2.1: Pohled na arény a haldy

2.3 Chunky

Chunk je část paměti, která může být alokována pomocí funkce `malloc()` a pak být uvolněna pomocí funkce `free()`. Chunk tedy může být buď v alo-

kovaném stavu nebo v uvolněném stavu. Každý chunk je součástí právě jedné haldy a patří do právě jedné arény.

Informace o každém chunku obsahuje struktura `malloc_chunk` (hlavička chunku) definovaná následovně:

```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;
  INTERNAL_SIZE_T      mchunk_size;

  struct malloc_chunk* fd;
  struct malloc_chunk* bk;

  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};
```

Výpis kódu 2.3: Hlavička chunku – struktura `malloc_chunk` (zdrojový soubor glibc v. 2.29 /malloc/malloc.c, ř. 1044-1055 kráceno)

Datový typ INTERNAL_SIZE_T je definován[5] (pokud není nastaven jinak) jako datový typ `size_t`, na 32bitových procesorech má velikost 4 byty, na 64bitových procesorech 8 bytů. Proměnná `mchunk_prev_size` obsahuje velikost předcházejícího chunku v bytech, pokud je předchozí chunk uvolněn, jinak může obsahovat uživatelská data z předchozího chunku. Proměnná `mchunk_size` obsahuje velikost tohoto chunku v bytech.

Ukazatele na následující chunk `*fd` a předcházející chunk `*bk` jsou použity pouze v případě, že je daný chunk uvolněn. V tom případě poté ukazují na následující a předchozí chunk v obousměrně zřetězeném spojovém seznamu uvolněných chunků. Pokud uvolněn není, v paměti jsou místo nich uložená uživatelská data.

Obdobně jsou použity ukazatele `*fd_nextsize` a `*bk_nextsize`. Ty jsou navíc použity pouze tehdy, pokud uvolněný chunk patří dle velikosti do velkého koše.

Jelikož je každý chunk zarovnán na 2 * sizeof(size_t) – tedy minimálně 8 bytů, můžou být 3 nejméně významné bity použity pro uložení následujících příznaků (řazeno od nejméně významného bitu).

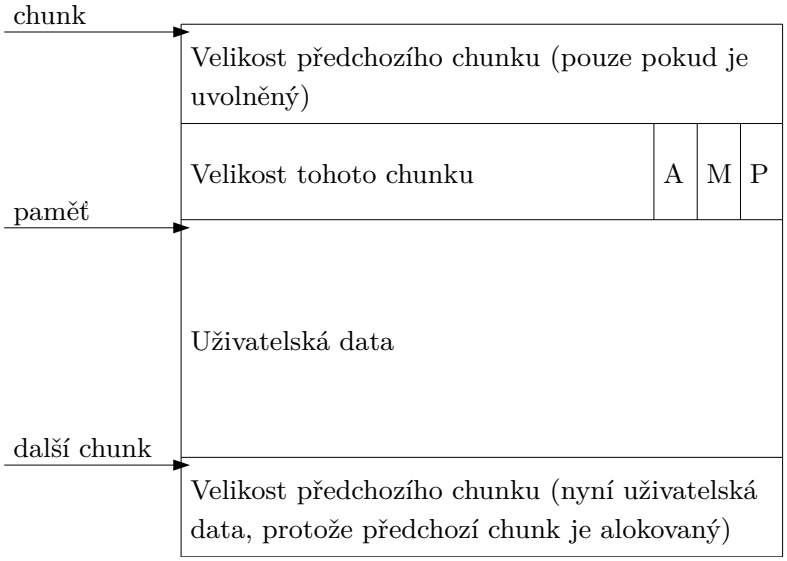
P (PREV_INUSE) bit Tento bit udává, zdali se předchozí chunk využívá. Pokud je nastaven, velikost předchozího chunku uložená v proměnné `mchunk_prev_size` není platná. Pokud není nastaven, předchozí chunk je uvolněný a lze dopočítat jeho předchozí adresu. První chunk má vždy tento bit nastaven, aby nedošlo k nesprávnému přístupu do paměti.

[5] zdrojový soubor glibc 2.29 /malloc/malloc-internal.h, řádek 55

M (IS_MMAPPED) bit Pokud je tento bit nastaven, chunk byl alokován pomocí funkce `mmap()`. Ostatní příznaky jsou v tomto případě ignorovány, protože chunk není součástí žádné arény/haldy.

A (NON_MAIN_ARENA) bit Nastavení bitu na hodnotu 1 udává, že chunk není alokován v rámci hlavní arény. Při nastavení na 0 se chunk nachází v hlavní aréně a na prvotní haldě.

Ukázka alokovaného chunku se nachází na obrázku 2.2, ukázka uvolněného chunku se nachází na obrázku 2.3. Ukazatel chunk představuje ukazatel, s kterým interně pracuje paměťový manažer. Ukazatel paměť představuje ukazatel, s kterým pracuje uživatel – získá jej při alokaci paměti jako návratovou hodnotou funkce `malloc()` a předává jej funkci `free()` pro uvolnění této paměti. Ukazatel další chunk představuje začátek dalšího chunku v paměti.



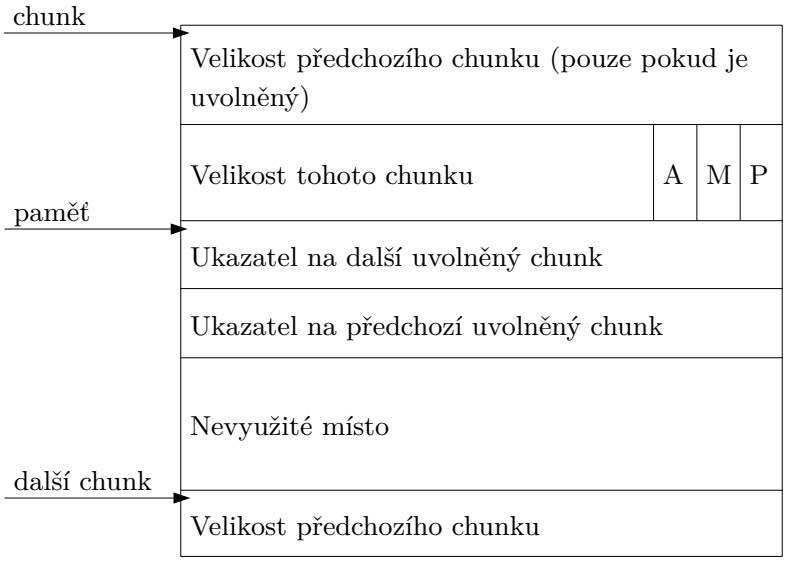
Obrázek 2.2: Pohled na alokovaný chunk v paměti

V kapitole číslo 2.1 Arény byly zmíněny ukazatele na 2 speciální případy chunků – top chunk a last remainder chunk.

Top chunk Každá aréna má svůj top chunk, ve struktuře `malloc_state` je na něj uložený ukazatel. Tento chunk se nachází na konci paměti dané arény a využívá se při alokaci paměti, když není dostupný volný blok paměti nalezen v koších (bins).

V případě, že požadovaný blok je menší než top chunk, je z top chunku vyčleněn požadovaný blok paměti, z kterého je vytvořen nový chunk a top chunk je zmenšen o tento díl.

Je-li požadovaný blok paměti větší než velikost top chunku, nastávají dvě možnosti. V případě hlavní arény je pomocí funkce `sbrk()` zvětšena



Obrázek 2.3: Pohled na uvolněný chunk v paměti

prvotní halda, tím je zvětšen top chunk, z kterého pak může být oddělen nový chunk. V případě ostatních arén je top chunk zvětšen pomocí funkce `mmap()`.

Last remainder chunk Obdobně jako na top chunk si každá aréna uchovává ukazatel na last remainder chunk. Když při alokaci chunku dojde k rozdělení většího chunku na požadovanou velikost a zbývající kus, tento poslední oddělený zbytek se nazývá last remainder chunk. Používá se pro zlepšení cache hit rate.

2.4 Koše (bins)

Koše slouží k ukládání uvolněných chunků, které jsou organizovány pomocí spojových seznamů. Na rozdíl od alokovaných chunků, které nejsou nijak sledovány, je třeba uvolněné chunky evidovat, aby se mohly znovu poskytnout k alokaci. Kvůli dosažení vyšší efektivity při prohledávání košů existují 4 kategorie rozdělené podle velikosti a funkčnosti. Koše jsou uloženy ve struktuře `malloc_state` (hlavička arény). Pole `fastbins[]` obsahuje ukazatele na rychlé koše, pole `bins[]` ukazatele na ostatní koše.

Fast bins Rychlé koše jsou určené pro malé uvolněné chunky. Jejich funkcí je zrychlení celého procesu dealokace-alokace pro malé kusy paměti. Každý rychlý koš má určenou velikost chunku, který přijímá. Chunky v rychlém koši jsou pouze jednosměrně zřetězené a nemění PREV_INUSE bit následujícího chunku. Chunk z rychlého koše může být časem přesunut do jiného koše.

Unsorted bin Nesetříděným košem je koš číslo 1 (ukazatel `bins[1]`). Tento koš obsahuje chunky různých velikostí. Uvolněný chunk (pokud není místo v tcache[6], nepatří do rychlého koše nebo není namapovaný pomocí funkce `mmap()`) bývá nejprve přidělen do nesetříděného koše a teprve časem se může dostat do malého/velkého koše.

Small bins Chunk velikosti menší než 512 bytů je označen jako malý a patří do malého koše. Tyto koše mají přesně určeno, pro jakou velikost chunku jsou stanoveny. Malých košů je 62 a jejich velikosti se zvětšují po 8 bytech. Nejmenší je koš číslo 2 (ukazatel `bins[2]`) pro chunky o velikosti 16 bytů, následuje koš číslo tři pro chunky o velikost 24 bytů atd.

Large bins Pro chunky o velikosti 512 bytů a větší existují velké koše. Tyto koše již nejsou určeny pro přesnou velikost, ale pro určitý interval velikosti. V daném koši jsou pak chunky řazeny od největších po nejmenší.

Celkově existuje 63 velkých košů, jejich rozdělení je odstupňované následovně:

- 32 košů, jejich velikost je odstupňována po 64 bytech (prvním velkým košem je koš číslo 65 pro velikost chunku 512–568 bytů);
- 16 košů s velikostí narůstající po 512 bytech;
- 8 košů s velikostí narůstající po 4 096 bytech;
- 4 košů s velikostí narůstající po 32 768 bytech;
- 2 košů s velikostí narůstající po 262 144 bytech;
- 1 koš pro zbývající velikosti chunků.

2.5 Thread local cache

Thread local cache (neboli tcache) je nový mechanismus zavedený do paměťového manažeru v glibc ve verzi 2.26. Má podobný účel jako koše. Cílem je zefektivnit správu uvolněných košů ve vícevláknových aplikacích.

Každé vlákno má vlastní tcache [12], která obsahuje několik chunků, s kterými je možné pracovat bez zamykání. Tím je docíleno značného zrychlení. Chunky jsou uloženy ve struktuře podobné rychlým košům. Pro konkrétní velikost chunku existuje konkrétní jednosměrně zřetězený seznam. Oproti rychlým košům je maximální velikost každého seznamu omezena.

[6] thread local cache

Kapitola 3

Známé útoky na haldu glibc

V této kapitole jsou popsány známé útoky na paměťový manažer knihovny glibc. K jejich provedení musí být program zranitelný na přetečení haldy (možnost ovlivnění vnitřních struktur haldy) nebo musí být špatně použity funkce pro správu paměti (`malloc()`, `free()`). Často je potřeba i jejich kombinace.

Cílem této kapitoly není ukázat provedení útoku na konkrétním zdrojovém kódu, ale popsat principy útoku. Výsledkem nemusí být vždy zápis na určité místo v paměti, ale také například ovlivnění ukazatele vráceného funkcí `malloc()`. V tom případě provede přepsání konkrétní adresy až zápis do „korektně“ alokovaného bloku paměti.

3.1 Unlink makro

První případ přepsání paměťových struktur haldy a tím možnosti ovlivnění paměťového manažeru byl popsán v roce 2000 [14]. V prohlížeči Netscape bylo možné způsobit přetečení haldy při dekódování jpeg obrázku. To mohlo vést k vykonání cizího kódu.

Důkladný popis tohoto možného napadení paměťového manažeru knihovny glibc (a paměťového manažeru pro System V malloc) byl sepsán o rok později v článku „*Once upon a free()*“ [15]. Pro tento způsob zranitelnosti se vžil název *unlink makro*.

Tento způsob útoku byl možný v době, kdy knihovna glibc neobsahovala žádné bezpečnostní kontroly. Aktuální verze se v některých kritických místech snaží kontrolovat, zdali nebyly pozměněny vnitřní struktury haldy. Proto již tento způsob útoku není možný.

Unlink makro je proveditelné na glibc verze 2.2.5. Vysvětlení bude vycházet ze zdrojových kódů této verze [16]. Pro vysvětlení útoku je potřeba znát, jakým způsobem probíhá dealokace paměti – jaký je algoritmus funkce `free()`[7].

[7] Funkce `free()` v současné verzi glibc funguje obdobně.

3.1.1 Dealokace paměti – funkce free()

Předchozím chunkem se myslí chunk ležící v paměti před uvolňovaným chunkem. Následujícím chunkem se myslí chunk ležící za uvolňovaným chunkem. Algoritmus funkce `free()` je následující:

1. Zavolání `free(0)` neudělá nic.
2. Pokud je chunk alokovaný pomocí funkce `mmap()`, bude uvolněn pomocí `munmap()`.
3. Pokud chunk přímo sousedí s top chunkem a:
   a) předchozí chunk je neuvolněn, je sloučen uvolňovaný chunk s top chunkem. Tím vznikne nový top chunk.
   b) předchozí chunk je uvolněn, tyto dva chunky jsou spojeny. Proto je nutné vyjmout z daného koše předchozí chunk a následně jej spojit s uvolněným chunkem. Potom je tento spojený chunk spojen s top chunkem a je nastaven nový top chunk.
4. Pokud je předchozí či následující chunk uvolněn, uvolňovaný chunk je s nimi sloučen. Pro toto sloučení je opět nutné jej vyjmout z odpovídajícího koše. Následně je sloučený chunk umístěn do odpovídajícího koše.

3.1.2 Vyjmutí chunku z koše

Jelikož jsou chunky v koších organizovány v obousměrně zřetězených seznamech, není vyjmutí chunku z koše triviální záležitostí. Pro vyjmutí je třeba „přepojit“ ukazatele mezi předchozím a následujícím chunkem v koši. Tuto operaci má na starosti unlink makro definované následovně:

```
#define unlink(P, BK, FD) { \
  BK = P->bk;               \
  FD = P->fd;               \
  FD->bk = BK;              \
  BK->fd = FD;              \
}
```

Výpis kódu 3.1: Unlink makro (zdrojový soubor glibc v. 2.2.5 /malloc/malloc.c, ř. 2516-2522)

Proměnná `P` představuje ukazatel na chunk, který je potřeba z koše vyjmout. Proměnná `BK` je ukazatel na předchozí chunk v koši a proměnná `FD` je ukazatel na následující chunk v koši. Makro si uloží ukazatel na předchozí a následující chunk v koši a následně přepíše u následujícího chunku ukazatel na předchozí chunk a opačně. Vyjme tedy prostřední chunk a propojí krajní.

3.1.3 Zranitelnost unlink makra

Proces vyjmutí chunku z koše je zranitelný. Paměťový manažer neověřuje, zda nebyly porušeny vnitřní struktury haldy. Pokud dojde k přetečení na haldě, může útočník upravit následující chunky či vytvořit falešný chunk.

Pokud bude mít uvolněný chunk, který je potřeba vyjmout z koše, změněné ukazatele na předchozí a následující chunk, dojde k nesprávnému zápisu do paměti. Přesněji stačí do ukazatele `P->fd` uložit adresu, kam se má zapisovat. Od této adresy je ještě nutné (na 32bitové architektuře) odečíst 12 bytů (velikost předcházejícího chunku, velikost tohoto chunku a ukazatel na další chunk) a poté do ukazatele `P->bk` uložit hodnotu zápisu.

Pro úspěšný útok stačí již jen vyvolat použití *unlink makra*. Toho lze dosáhnout vhodnou manipulací s chunky. Možný způsob, jakým lze útok provést, bude popsán v kapitole číslo 4.1.

3.1.4 Oprava zranitelnosti

V současné době již tento způsob útoku na knihovnu glibc není možný. Ve verzi 2.3.4 byla provedena úprava[8] unlink makra, která ověřuje, zda nebyl poškozen daný spojový seznam.

Oprava přidává kontrolu, zdali `FD->bk == P` a `BK->fd == P`. Pokud následující a předcházející chunk správně neukazuje na chunk, který se má vyjmout, je vypsána chyba o poškození spojového seznamu a běh programu je ukončen.

3.2 Double free

K uvolnění paměťového bloku slouží funkce `free()`. Volání této funkce na již uvolněný blok paměti způsobí nedefinované chování [17]. Tato programátorská chyba může vést až k zápisu na libovolné místo v paměti [18].

Při prvním zavolání funkce `free()` na alokovaný blok paměti (chunk) dojde k jeho změně. Tento chunk změní svůj stav z alokovaného chunku (obrázek 2.2) na uvolněný chunk (obrázek 2.3). Místo uživatelských dat se nachází ukazatele na předchozí a následující uvolněný chunk v příslušném koši a zbytek prostoru je nevyužitý. Dále je u následujícího (myšleno umístěním v paměti) chunku nastavena velikost předchozího chunku a vynulován `PREV_INUSE` bit.

Bezpečným jednáním může být změna ukazatele na tuto paměť po jejím uvolněním na hodnotu `NULL`. S pamětí se nesmí dále pracovat. Tímto způsobem je zamezeno zápisu do dané paměti a případné opětovné volání funkce `free()` nic neprovede.

Pokud dojde k opakovanému uvolnění, může se stát cokoliv. Již uvolněná paměť mohla být opět přidělena (mezi prvním a druhým uvolněním). Potenciální útočník mohl legitimně změnit tuto paměť (zápisem do nově alokovaného chunku). Vhodnou úpravou mohl vytvořit falešné chunky. A právě

[8] commit 3e030bd5f9fa57f79a509565b5de6a1c0360d953

tento falešný ovlivněný chunk se může nacházet na adrese, která bude podruhé uvolněna. Vhodnou přípravou falešných chunků může dojít ke spuštění *unlink makra* a paměťový manažer může provést zápis na takřka libovolnou adresu. Možný způsob útoku může vypadat následovně [18]:

1. Na haldě jsou za sebou alokované bloky A a B. Nejdříve je uvolněn blok A, poté je uvolněn blok B. Ukazatel na blok B není po uvolnění nastaven na `NULL` a bude uvolněn (v bodě 4) podruhé. Během prvního uvolnění bloku B dojde ke sloučení s blokem A a tento celek je uložen do koše.
2. Je alokován blok paměti o velikosti $A + B$. Pro tento blok je vybrán chunk z koše z předchozího bodu, tedy alokovaný bod se rozprostírá přes paměť předchozích bloků A a B.
3. Do nově alokované paměti je provedený regulérní zápis. Je vytvořen falešný chunk na místě těsně před původním chunkem B. U falešného chunku je potřeba nastavit jeho ukazatel `*fd` na adresu (mínus 12 bytů), kde bude proveden zápis a ukazatel `*bk` na hodnotu, která má být zapsána (podrobněji kapitola 3.1 Unlink makro).
4. Chunk B je uvolněn podruhé. Při jeho uvolnění je sloučen s předchozím falešným chunkem vytvořeným v bodě 3. Během sloučení je volané *unlink makro*, paměťový manažer provede zápis do paměti dle ukazatelů.

Tento klasický způsob exploitace *double free* již není možný díky bezpečnostním kontrolám v *unlink makru*. Nejedná se o jediný možný způsob zneužití. Opakovaným uvolněním může například dojít k poškození struktury haldy tak, že následná alokace paměti může opakovaně vracet stejný blok paměti.

3.3 Use after free

MITRE [19] popisuje tuto zranitelnost jako práci s dynamicky alokovaným kusem paměti poté, co byl uvolněn. V momentě, kdy jednou uvolníme dynamicky alokovaný kus paměti, nesmíme již s touto pamětí pracovat. Ukázku možného *use after free* zobrazuje následující příklad:

```
char *buffer = malloc(size);
if(clear) {
    free(buffer);
}
// [...]
if(error) {
    printf("%s", buffer);
}
```

Výpis kódu 3.2: Ukázka zranitelnosti use after free

Pokud během vykonávání programu byla splněna podmínka `clear`, byl alokovaný blok uvolněn. Ukazatel `*buffer` ale stále ukazuje do dané paměti. Uvolněním nedošlo k „zneplatnění“ ukazatele. Pokud poté bude splněna podmínka `error`, dojde ke čtení z adresy, ve které může být cokoliv. Tato paměť již programu nepatří.

Způsob exploitace zranitelnosti závisí na konkrétním programu. Tato chyba může vést k přepsání jiných dat programu, k odhalení umístění glibc, haldy a také ke spuštění kódu. Možným způsobem obrany může být nastavení ukazatele na uvolněnou paměť po uvolnění na hodnotu `NULL`.

3.4 The Malloc Maleficarum

Nový pohled na možnosti exploitace haldy přinesl v roce 2005 příspěvek do e-mailové konference Bugtraq nazvaný *The Malloc Maleficarum* [20]. Po sérii oprav knihovny glibc v roce 2004 byly známé techniky útočení na haldu zastaralé. Přidané kontroly v kódu znemožňovaly útok. Malloc Maleficarium přinesl nové techniky útoku pojmenované:

- The House of Prime;
- The House of Mind;
- The House of Force;
- The House of Lore;
- The House of Spirit.

Tento příspěvek obsahoval pouze teoretický popis útoků. Neukázal žádnou praktickou ukázku. Za „*praktického průvodce Malloc Maleficarium*“ se označuje článek do časopisu Phrack *Malloc Des-Maleficarum* vydaný až o 4 roky později, tedy v roce 2009 [21]. Tento článek ukázal možnosti útoku na konkrétních ukázkách kódu. Následující popis možných technik vychází z těchto dvou článků a ukazuje princip útoku na zdrojových kódech glibc ve verzi 2.3.5.

3.4.1 House of Prime

House of Prime je jednou z nejvíce komplikovaných technik popsaných v Malloc Maleficarium. Dle Phantasmal Phantasmagoria [20] se jedná o nejméně užitečnou techniku, sám doporučuje použít raději *House of Spirit*, pokud je to možné.

Cílem House of Prime je donutit paměťový manažer, aby při alokaci přidělil blok na ovlivněné adrese – ideálně například blok alokovaný na zásobníku. Regulérním zápisem do přidělené paměti dochází k přepsání cíle útoku. Postup se dá shrnout do následujících kroků:

1. Přetečením je přepsána velikost dvou chunků.
2. Vytvoření falešné arény v druhém chunku, jejíž koše ukazují na cíl přepsání.
3. Uvolněním prvního chunku dojde k přepsání `av->max_fast`.
4. Uvolněním druhého chunku dojde ke změně `arena_key`.
5. Volání funkce `malloc()` vrátí ukazatel na požadovanou adresu.

Nejprve je nutné změnit proměnnou `av->max_fast`. Ta udává maximální velikost rychlého koše. Kvůli dalšímu postupu je nutné ji změnit na nějakou velkou hodnotu. K tomu se využije chyba v části kódu funkce `free()`, která zajišťuje vložení uvolňovaného chunku do rychlého koše.

```
if ((unsigned long)(size) <= (unsigned long)(av->max_fast)) {
    // [...]
    set_fastchunks(av);
    fb = &(av->fastbins[fastbin_index(size)]);
    // [...]
    p->fd = *fb;
    *fb = p;
}
```

Výpis kódu 3.3: Část funkce `free()` zajišťující vložení chunku do rychlého koše (zdrojový soubor glibc v. 2.3.5 /malloc/malloc.c, ř. 4244-4274 kráceno)

Tento kód je vykonán, pokud velikost uvolňovaného chunku je menší nebo rovna velikosti největšího rychlého koše. Problém může nastat při určování, do kterého rychlého koše má být chunk zařazen. K tomuto určení slouží makro `fastbin_index()` definované následovně:

```
#define fastbin_index(sz) ((((unsigned int)(sz)) >> 3) - 2)
```

Výpis kódu 3.4: Makro pro výpočet indexu rychlého koše (zdrojový soubor glibc v. 2.3.5 /malloc/malloc.c, ř. 2118)

Minimální velikost korektního chunku je 16 bytů. Tento chunk by byl správně zařazen do koše číslo 0. Pokud ale útočník změní velikost chunku na 8 bytů, makro vrátí hodnotu -1. Do ukazatele fb bude uložena adresa `&(av->fastbins[-1])`. Dle definice struktury `malloc_state` by měla být v paměti před polem `fastbins[]` uložená proměnná `max_size`. Ta bude následně změněna na hodnotu rovnající se adrese chunku.

Dalším krokem je přepsání `arena_key`. Tato proměnná je ukazatelem na danou arénu konkrétního vlákna. Proměnná by se měla nacházet v paměti

za strukturou arény. Pokud je před ní, není House of Prime použitelný. Kvůli přepsání `av->max_fast` lze obdobným způsobem kód zneužít k zápisu daleko za pole rychlých košů – stačí uvolnit chunk dostatečné velikosti. Uvolněním druhého chunku s upravenou velikostí dojde k přepsání `arena_key` na adresu druhého uvolňovaného chunku. Z chunku se stala aréna.

Aby volání funkce `malloc()` vrátilo ovlivněnou adresu, musí být v druhém chunku správně vytvořená falešná aréna. Pokud je cílem přesměrování toku programu, jedním možným způsobem je, aby funkce `malloc()` vrátila paměť na zásobníku. Následně pak lze zápisem do tohoto bloku paměti přepsat návratovou adresu.

Ve funkci `malloc()` lze zneužít kód pro alokování chunku z rychlých košů. To má na starosti následující část kódu:

```
if ((unsigned long)(nb) <= (unsigned long)(av->max_fast)) {
  long int idx = fastbin_index(nb);
  fb = &(av->fastbins[idx]);
  if ( (victim = *fb) != 0) {
    if (__builtin_expect (fastbin_index (chunksize (victim)) !=
    ↪  idx, 0))
      malloc_printerr (check_action, "malloc(): memory
      ↪  corruption (fast)", chunk2mem (victim));
    *fb = victim->fd;
    check_remalloced_chunk(av, victim, nb);
    return chunk2mem(victim);
  }
}
```

Výpis kódu 3.5: Část funkce `malloc()` zajišťující alokaci chunku z rychlého koše (zdrojový soubor glibc v. 2.3.5 /malloc/malloc.c, ř. 3853-3864)

Aby byly prohledány rychlé koše, musí být velikost žádaného chunku menší nebo rovna `av->max_fast`. Ukazatel `av` ukazuje na falešnou arénu – druhý uvolněný chunk, tudíž `av->max_fast` lze dle potřeby nastavit. Žádaný chunk tudíž může být mnohem větší velikosti, než je standardní největší velikost pro rychlý koš.

Poslední nutnou podmínkou je splnění bezpečnostní kontroly, zdali chunk v koši má stejnou velikost, jako je velikost koše. Pokud je cílem získat chunk na zásobníku, je třeba mít možnost do něj zapsat správnou velikost chunku a použít adresu mínus 4 byty (32bitová architektura) od této velikosti.

Následně je podle požadované velikosti vybraný adekvátní rychlý koš a jeho adresa je vrácena funkcí `malloc`. Nejjednodušším řešením je naplnit druhý chunk adresou, kterou je třeba získat (adresa na zásobníku, kde je uložena velikost mínus 4 byty).

V současnosti již tento způsob útoku není možný, byl velice rychle opraven[9]. Do makra pro výpočet správného koše byla přidána kontrola na minimální velikost chunku. Knihovna glibc obsahuje tuto opravu od verze 2.4. Dále již není v aktualních verzích knihovny používána proměnná `av->max_fast`, nahradilo ji makro `MAX_FAST_SIZE`.

3.4.2 House of Mind

Druhou popsanou technikou je House of Mind. Ten se podobá původnímu *unlink makru* (kapitola 4.1). K přepsání části paměti je potřeba pouze jedno volání funkce `free()` na blok paměti, který byl změněn pomocí přetečení. Zápis provede část kódu pro organizaci nesetříděných košů. Celkový postup se dá shrnout do následujících kroků:

1. Pomocí přetečení je změněn `NON_MAIN_ARENA` bit na 1 u chunku, který bude (v bodě 4) uvolněn.

2. Je proveden zápis na adresu, na které by se měl nacházet ukazatel `heap_info->ar_ptr` pro chunk z bodu 1.

3. Na adrese, kam ukazuje ukazatel z bodu 2, je vytvořena falešná aréna.

4. Je uvolněn chunk z bodu 1.

Základem útoku je přepsání příznaku `NON_MAIN_ARENA` na 1 u chunku, který bude uvolněn. Pokud je tento příznak nastaven, znamená to, že se chunk nenachází v hlavní aréně. Po zavolání funkce `free()` se zjišťuje aréna pro uvolňovaný chunk. To mají na starosti následující makra:

```
#define HEAP_MAX_SIZE (1024*1024)
#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)
#define heap_for_ptr(ptr) \
    ((heap_info *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))
#define arena_for_chunk(ptr) \
  (chunk_non_main_arena(ptr) ? heap_for_ptr(ptr)->ar_ptr :
↪  &main_arena)
```

Výpis kódu 3.6: Makra sloužící k určení arény chunku (zdrojové soubory glibc v. 2.3.5 /malloc/malloc.c a /malloc/arena.c)

Nejprve je zavoláno makro `arena_for_chunk(ptr)`. Jelikož je nastaven příznak `NON_MAIN_ARENA`, je aréna chunku zjišťována z hlavičky haldy (struktura `heap_info`) pomocí makra `heap_for_ptr(ptr)`. Hlavička se nachází na začátku haldy – pro chunky mimo hlavní arénu se vypočte z jejich adresy.

[9] commit bf58906631af8fe0d57625988b1d003cc09ef01d

Pokud by byl takto přepsaný chunk na začátku haldy – například na adrese `0x08049600`, jeho „hypotetická“ hlavička haldy by se nacházela na adrese `0x08000000`. Tato adresa je níže než halda, nelze na ni zapisovat. Pokud ale chunk bude na vyšší adrese – například `0x08100100`, bude hlavička na adrese `0x08100000`, která se nachází v oblasti haldy, a útočník ji může podvrhnout. Na tuto adresu je tedy potřeba uložit adresu falešné arény.

Dalším krokem je vytvoření této arény. Pro její správné nastavení je třeba vysvětlit další průběh útoku. Během uvolnění ovlivněného chunku je nejdřív zavolána funkce `public_fREe(Void_t* mem)`, v ní je zjištěna aréna pro daný chunk a je zavolána funkce `_int_free(ar_ptr, mem)`. Je nutné překonat několik bezpečnostních kontrol a vyvolat část kódu, která uvolněný chunk přidá do nesetříděného koše. Proto je nutné splnit několik podmínek. Tím, že je aréna pod kontrolou, není problém vhodné podmínky zařídit. Zejména je nutné zařídit následující:

- chunk nesmí mít nastavený příznak `IS_MMAPPED`;
- chunk musí mít nastavený příznak `PREV_INUSE`;
- chunk musí mít větší velikost než `av->max_fast`;
- další chunk není top chunk (`av->top`);
- další chunk byl korektně alokován;
- chunk za dalším chunkem má nastavený příznak `PREV_INUSE`;
- `av->max_fast` musí mít nastavený příznak `NONCONTIGUOUS_BIT`[10].

Pokud vše proběhlo správně, dojde při uvolnění bloku k vykonání kódu sloužícímu k zařazení chunku do nesetříděného koše:

```
bck = unsorted_chunks(av);
fwd = bck->fd;
p->bk = bck;
p->fd = fwd;
bck->fd = p;
fwd->bk = p;
```

Výpis kódu 3.7: Část funkce `free()` zajišťující zařazení chunku do nesetříděného koše (zdrojový soubor glibc v. 2.3.5 /malloc/malloc.c, ř. 4338-4343)

Proměnné `bck` a `fwd` jsou dočasné proměnné, `p` je ukazatel na uvolňovaný chunk. Do `bck` bude makrem uložena adresa koše (`&av->bins[0]`). Následně je do `fwd` uložena hodnota uložená na adrese `fwd+8` (32bitová architektura).

[10] `av->top = av->top | 2;`

Posledním řádkem je na adresu `fwd+12` (neboli `*(&av->bins[0]+8)+12`) zapsána adresa chunku.

Pokud bude ve falešné aréně na adrese `&av->bins[0]+8` uložena návratová adresa mínus 12 bytů, bude tok programu přesměrován na adresu chunku – tedy adresa bloku předaného k uvolnění mínus 8 bytů. Dle Blackngela [21] je vhodné pro úspěšný útok zapsat do `p->prev_size` instrukci `jump 0x0c`, dále prvních 16 bytů bloku změnit na instrukci `nop` a do zbytku umístit kód k vykonání.

Použití techniky House of Mind bylo zabráněno přidáním[11] bezpečnostní kontroly do kódu pro nesetříděné koše ve verzi 2.11.

3.4.3 House of Force

Tato technika se zabývá exploitací *wilderness* neboli top chunku. Ten má speciální funkci při alokaci paměti. Má stejnou strukturu jako obyčejný chunk, avšak může se zvětšovat či zmenšovat a nachází se vždy na konci paměťového prostoru dané arény. Tento chunk není nikdy uvolněn a neobsahuje uživatelská data.

House of Force mění velikost top chunku a tím umožňuje alokaci abnormálně velkého bloku paměti. Následné další volání funkce `malloc()` může vrátit blok paměti umístěný třeba až na zásobníku a zápisem do tohoto bloku lze přepsat návratovou adresu. Provedení útoku se dá shrnout následujícími kroky:

1. Přepsání velikosti topchunku pomocí přetečení.
2. Volání funkce `malloc()` s ovlivnitelnou velikostí.
3. Další volání funkce `malloc()` s možností zápisu do nově alokované paměti.

Prvním krokem je přepsání velikosti top chunku pomocí přetečení. Velikost je vhodné změnit na velké číslo, na 32bitové architektuře ideálně 0xffffffff. Tato velikost musí být větší než velikost bloku alokovaného v kroku 2.

Druhým krokem je posunutí ukazatele na top chunk (`av->top`). Toho se dosáhne voláním funkce `malloc()`. Pro provedení útoku musí mít útočník pod kontrolou velikost alokovaného bloku. Kvůli změně velikosti top chunku v předchozím kroku bude uspokojen požadavek na alokaci bloku paměti právě z něj. Tím zároveň dojde k posunutí top chunku dále do paměti. Toto chování vyvolává následující část kódu z funkce `malloc()`:

```
use_top:
victim = av->top;
size = chunksize(victim);
```

[11] commit f6887a0d9a55f5c80c567d9cb153c1c6582410f9

```
if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
  remainder_size = size - nb;
  remainder = chunk_at_offset(victim, nb);
  av->top = remainder;
  set_head(victim, nb | PREV_INUSE |
           (av != &main_arena ? NON_MAIN_ARENA : 0));
  set_head(remainder, remainder_size | PREV_INUSE);

  check_malloced_chunk(av, victim, nb);
  return chunk2mem(victim);
}
```

Výpis kódu 3.8: Část funkce `malloc()` zajišťující alokaci chunku z top chunku (zdrojový soubor glibc v. 2.3.5 /malloc/malloc.c, ř. 4151-4181 kráceno)

Jako `victim` (potenciální volná paměť) je označený top chunk a je spočítána jeho velikost. Pokud je žádaná velikost (`nb`) menší než velikost top chunku (`size`), může být top chunk rozdělený na dva bloky. V prvním bloku bude vytvořen alokovaný chunk. Z druhého bloku se stane nový top chunk.

Pokud by byla cílem útoku například návratová adresa, alokací velmi velkého bloku paměti dojde k posunutí top chunku třeba až na zásobník. Posledním krokem je druhé volání funkce `malloc()`, které vrátí paměť začínající na adrese `av->top`. Regulérním zápisem do této paměti lze následně návratovou adresu přepsat.

House of Force bylo dlouhou dobu proveditelné. Provedení je možné ještě ve verzi glibc 2.28. Až v nejnovější verzi glibc 2.29 je chyba opravena[12].

3.4.4 House of Lore

House of Lore zneužívá fungování košů. Cílem je vložení falešného chunku do malého nebo velkého koše. Při alokaci paměti je následně vrácen funkcí `malloc()` blok na ovlivněné adrese. Regulérním zápisem do tohoto alokovaného bloku lze provést zápis na libovolné místo v paměti.

Tato technika nebyla vysvětlena na praktické ukázce ani v *Malloc Maleficarum* ani v *Malloc Des-Maleficarum* [20, 21]. Názornou ukázku přinesl článek *The House Of Lore: Reloaded* [22], z něj bude vycházet následující popis.

V této práci bude popsán pouze útok na malé koše. Zneužití velkých košů bylo brzy zabráněno výměnou[13] zranitelného kódu za unlink makro ve verzi glibc 2.6. Pro provedení útoku pomocí malých košů jsou nutné následující kroky:

1. Alokovaný chunk malé velikosti je uvolněn.

[12] commit 30a17d8c95fbfb15c52d1115803b63aaa73a285c
[13] commit 7ecfbd386a340b52b6491f47fcf37f236cc5eaf1

2. Alokovaný další chunk větší velikosti než předchozí je uvolněn.
3. Pomocí přetečení je uvolněný chunk změněn.
4. Chunk stejné velikosti jako uvolněný chunk je několikrát alokovaný.
5. Do alokovaného chunku je proveden zápis.

Na začátku musí být alokovaný chunk, který bude následně uvolněn a velikostí bude spadat do malého koše. Další podmínkou je, že pomocí přetečení je možné (později) přepsat jeho ukazatel `*bk`. V momentě, kdy bude na tento chunk zavolána funkce `free()`, bude přesunut do nesetříděného koše.

Dalším krokem je přesunutí chunku z nesetříděného koše do odpovídajícího malého koše. Toho je dosaženo pomocí dalšího volání funkce `malloc()`. Je nutné, aby byl požadavek větší, než je velikost chunku, který má být přesunut do malého koše. Pokud by byl požadavek menší, mohlo by dojít k uspokojení požadavku právě z tohoto chunku.

Poté, co je chunk umístěn do malého koše, může dojít k přepsání ukazatele `*bk` chunku na adresu, kterou má vrátit volání `malloc()`. Chunky jsou v rychlém koši spojené v obousměrně zřetězeném seznamu. Nově přidaný chunk je přidaný na začátek seznamu, chunky pro alokaci jsou odebírány od konce. Z tohoto důvodu může být nutné provést několik volání funkce `malloc()` odpovídající velikosti, aby z koše byly vyjmuty další chunky a ovlivněný chunk se dostal na konec seznamu – tedy další požadavek na alokaci odpovídající velikosti jej vyjme z koše. Toto vyjmutí z koše má na starosti následující kód:

```
bck = victim->bk;
set_inuse_bit_at_offset(victim, nb);
bin->bk = bck;
bck->fd = bin;
```

Výpis kódu 3.9: Část funkce `malloc()` zajišťující alokaci chunku z malého koše(zdrojový soubor glibc v. 2.3.5 /malloc/malloc.c, ř. 3882-3885 kráceno)

Proměnná `victim` je ukazatel na blok v koši, který má být vyjmut. V proměnné `victim->bk` se tedy nachází pomocí přetečení přepsaný ukazatel. Ten je uložený do proměnné `bck`. Důležitým detailem je, že dojde k zápisu adresy koše na `bck->fd`, proto tato adresa musí být zapisovatelná. Tímto došlo k vložení „falešného chunku“ do rychlého koše, další volání funkce `malloc()` stejné velikosti vrátí chunk nacházející se na ovlivněné adrese.

Následující postup je obdobný, korektním zápisem do nově alokované paměti dojde k přepsání cíle útoku – např. návratové adresy.

Útok pomocí malých košů by měl být možný až do verze glibc 2.25. Avšak v průběhu let byly přidány bezpečnostní kontroly, které ověřují, zda není rozbitý obousměrně zřetězený seznam v malém koši. Kvůli nim je útok ztížený, ale není nemožný. Od verze glibc 2.26 útoku brání zavedení tcache.

3.4.5 House of Spirit

House of Spirit je odlišný od ostatních útoků. U této techniky se pomocí přetečení přepisuje ukazatel na alokovaný blok paměti. Následně, když je tento přepsaný ukazatel předaný funkci `free()`, dojde k uvolnění falešného bloku paměti. Tento uvolněný blok (vytvořený například na zásobníku) je uložen do rychlého koše. Další zavolání funkce `malloc()` odpovídající velikosti vrátí tento blok a regulérním zápisem do paměti může dojít k přepsání například návratové adresy.

Prvním krokem je tedy přepsání ukazatele na paměť, který bude následně předaný funkci `free()` k uvolnění. Je jedno, jestli se tento ukazatel nachází na zásobníku či haldě. Adresa, na kterou má být ukazatel změněn, se musí nacházet maximálně 64 bytů (maximální velikost chunku spadajícího do rychlého koše) před adresou, kterou chce útočník změnit (například návratová adresa).

Aby mohl být falešný chunk uložen do rychlého koše, musí být splněny tyto podmínky. Musí mít nastavenou velikost chunku na hodnotu menší než 80 bytů a další chunk musí mít nastavenou velikost větší než 2∗`size_t` a menší než `av->system_mem`.

Pokud byly tyto podmínky splněny, dojde k uložení falešného chunku do rychlého koše. Následná žádost o alokaci paměti stejné velikosti, jakou má falešný chunk, bude uspokojena tímto chunkem z rychlého koše. Regulérním zápisem do nově alokovaného bloku může dojít k přepsání cíle útočníkem.

Tato chyba nebyla v knihovně glibc nikdy opravena. Její provedení je možné na aktuální verzi knihovny glibc 2.29. Dokonce od verze 2.26 je použití *House of Spirit* ještě jednodušší [23]. Falešný chunk není nutné ukládat do rychlých košů, ale je možné jej uložit do tcache. Při této metodě navíc není nutné nastavovat velikost následujícího chunku.

3.5 House of Einherjar

House of Einherjar je moderní technikou představenou v roce 2016 [24]. Zneužívá mechanismus spojování dvou uvolněných chunků umístěných v paměti vedle sebe do jednoho. Pro provedení úspěšného útoku může stačit přetečení o pouhý jeden byte (*off-by-one overflow*). Výsledkem je ovlivnění ukazatele vráceného funkcí `malloc()`. Postup útoku je následující:

1. Pomocí přetečení je u chunku změněna velikost předchozího chunku (`prev_size`) a `PREV_INUSE` bit.

2. Je vytvořena hlavička falešného chunku v místě, které má být změněno (na zásobníku).

3. Pozměněný chunk s upravenou hlavičkou je uvolněn.

4. Je alokována paměť, do této paměti je proveden zápis.

Nechť jsou na haldě umístěny 2 chunky (za sebou). První chunk je zranitelný na přetečení. Druhý chunk je takové velikosti, aby nemohl být po uvolnění umístěn do rychlého koše. Regulérním zápisem na konci prvního chunku je změněna v druhém chunku velikost předchozího chunku[14]. Pomocí přetečení o jeden byte je vynulován příznak `PREV_INUSE` druhého chunku.

Dalším krokem je vytvoření falešného chunku před adresou, kterou má vrátit volání `malloc()`. Umístěn může být například na zásobníku. Falešný chunk musí mít nastaveny ukazatele na předcházející a následující chunk tak, aby proběhla správně bezpečnostní kontrola v unlink makru.

Nyní musí být druhý chunk uvolněn. Během uvolňování je druhý chunk spojen s falešným chunkem. Poté pokud je následujícím chunkem top chunk, je adresa falešného chunku nastavena jako adresa top chunku. Jinak je falešný chunk umístěn do nesetříděného koše. Po tomto uvolnění může být ještě nutné upravit velikost chunku na korektní[15] velikost – větší nebo rovno $2 * $ `SIZE_SZ` a menší než `av->system_mem`. Další volání funkce `malloc()` vrátí tento falešný chunk. Zápisem do této paměti je útok proveden.

Knihovna glibc verze 2.26 přinesla podporu thread local cache, která znemožňuje útok tímto způsobem. Během uvolnění nedochází ke sloučení s předchozím či následujícím blokem, ale chunk je rovnou vložen do tcache.

[14]Tato paměť se může překrývat. Pokud je chunk uvolněn, následující chunk má nastavenou jeho velikost v proměnné `prev_size`. Pokud chunk uvolněn není, jsou v tomto místě jeho data.

[15]zdrojový soubor glibc 2.25 /malloc/malloc.c, řádek 3472

Kapitola 4

Demonstrace útoků na glibc

V této kapitole bude popsáno praktické provedení vybraných útoků na zranitelných programech. Tato bakalářské práce se nezabývá způsobem překonání různých ochran systému. Pro zjednodušení postupu se vždy předpokládá minimálně vypnutí ASLR[16].

Pro prezentaci útoků bylo připraveno virtuální prostředí. Jedná se o základní instalaci operačního systému Ubuntu 18.04.2 LTS (64-bit) ve virtualizačním nástroji VirtualBox. Na přiloženém disku se nachází exportovaná appliance připravená pro import do vlastní instalace tohoto nástroje. V adresáři `~/Desktop` tohoto virtuálního prostředí jsou nachystány zdrojové kódy zranitelných programů, přeložené spustitelné soubory a skript k sestavení vstupu pro program.

Jelikož útoky mohou vyžadovat použití specifické (staré) verze glibc, jsou zdrojové kódy zranitelných programů přeložené a slinkované staticky s danou verzí glibc. Podrobnosti ohledně kompilace jsou u každé ukázky uvedeny zvlášť. Takto sestavený (staticky slinkovaný) program lze spustit i na moderní distribuci systému GNU/Linux a prezentovat na něm danou zranitelnost.

4.1 Unlink makro

Pro provedení tradičního útoku – Unlink makro – je nutná stará verze knihovny glibc. Zranitelný zdrojový kód v souboru `unlink_makro.c` byl zkompilován[17] na dobové distribuci Debian 3.0r6 (32-bit), verze kompilátoru gcc 2.95.4, slinkován staticky s knihovnou glibc verze 2.2.5. Pro zjednodušení byly přiloženy ladící informace. Útok byl otestován na takto sestaveném programu na distribuci Ubuntu 18.04.2 LTS (dále jen virtuální prostředí).

Ve zranitelném programu dochází k alokaci 2 paměťových bloků. Tyto bloky – chunk A a chunk B – se nacházejí v paměti přímo za sebou. Požadovaná

[16] `# sysctl -w kernel.randomize_va_space=0`

[17] `$ gcc -g -static unlink_makro.c -o unlink_makro-static-2-2-5`

velikost byla u každého 80 bytů, tedy celková velikost každého chunku je 88 bytů (jde o 32bitový program). Poté dojde ke čtení 100 bytů ze standardního vstupu do chunku A. Tímto načtením může dojít k přepsání hlavičky a části dat chunku B. Závěrem dojde k uvolnění paměti, nejprve je uvolněn chunk A a poté B.

Funkce `challenge()` by neměla být nikdy zavolána. Na spuštění této funkce bude prezentováno úspěšné provedení útoku. Pro něj je potřeba připravit vhodný vstup. Ten je zachycen v souboru `exploit.in`, který byl vygenerovaný perlovým skriptem v `exploit.pl`.

Cílem útoku je vyvolat unlink makro při uvolňování chunku A. Aby k tomu došlo, musí být předchozí nebo následující chunk v paměti označený jako uvolněný. V tomto případě je na to použit následující chunk B.

Informaci o tom, jestli je chunk B uvolněn, nese bit `PREV_INUSE` dalšího chunku za chunkem B. Ten je vypočítán tak, že se použije adresa chunku B a přičte se k ní velikost chunku B. Pokud se pomocí přetečení změní velikost chunku na -4 neboli 0xfffffffc, umístí se další chunk 4 byty před chunk B. Tedy `PREV_INUSE` bit označující uvolnění chunku B je vyhodnocen z adresy `chunkB->prev_size`. Ten stačí nastavit na 0. V řešení je celá tato proměnná změněna na 0xffffff08.

Tedy během uvolnění chunku A je zjištěno, že následující chunk v paměti je uvolněn, je jej tedy potřeba vyjmout z koše pomocí unlink makra. Nyní je potřeba zmanipulovat ukazatele `chunkB->fd` a `chunkB->bk`. Do ukazatele `*fd` se uloží adresa cíle přepsání mínus 12 bytů a do ukazatele `*bk` hodnota, která se má zapsat.

Problémem je, že volání funkce `free()` na chunk B způsobí z důvodu nesmyslné velikosti tohoto chunku pád programu. To lze vyřešit přesměrováním toku programu při zavolání právě této funkce. U dynamicky slinkovaného programu toho lze docílit změnou v *global-offset-table*, tato sekce se ovšem u staticky slinkovaného programu nenachází. Dle [14] lze ovšem pro přesměrování použít ukazatel `__free_hook`. Ten slouží k ladícím účelům, pokud je přepsán, tak po volání funkce `free()` je zavolán kód na této adrese.

Druhým problémem je vedlejší efekt vykonání unlink makra. Kromě zápisu do `chunkB->fd->bk` dojde následně k zápisu do `chunkB->bk->fd`. Tedy paměť 8 bytů za hodnotou, která má být zapsána, musí být zapisovatelná. To vylučuje přímé použití adresy funkce `free()`, jelikož tato paměť není zapisovatelná.

Řešením je přesměrovat tok programu do chunku A, odkud je pomocí 2 instrukcí přesměrován do funkce `challenge()`. Je nutné vynechat prvních 8 bytů, kde se po uvolnění uloží ukazatele na `*fd` a `*bk`. Pro zavolání je potřeba nejprve uložit adresu funkce `challenge()` na zásobník pomocí instrukce `push` a poté ji zavolat pomocí instrukce `ret`.

Tabulka 4.1: Struktura exploitu unlink makra

chunk A	8 bytů výplně
...	instrukce `push <challenge()>` a `ret;` (16 bytů);
...	64 bytů výplně
`chunkB->prev_size`	sudé číslo (vynulovaný `PREV_INUSE` bit) = 0xfffffff08
`chunkB->size`	-4 = 0xfffffffc
`chunkB->fd`	cíl přepsání - 12 = `__free_hook - 12`
`chunkB->bk`	hodnota přepsání = chunkA + 8

4.1.1 Obrana

Do knihovny glibc ve verzi 2.3.4 byla přidána ochrana, která ověřuje správnost obousměrně zřetězeného seznamu v koši. Předchozí i následující chunk musí správně ukazovat na chunk, který má být vyjmut. Toto ověření znemožňuje provedení útoku, neboť poškození spojového seznamu je jeho základem.

```
if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
  malloc_printf_nc (check_action, "corrupted double-linked list
  ↪  at %p!\n", P);
```

Výpis kódu 4.1: Bezpečnostní kontrola v unlink makru (zdrojový soubor glibc v. 2.3.4 /malloc/malloc.c, ř. 1986)

4.2 House of Einherjar

House of Einherjar je proveditelný na knihovně glibc bez zapnuté tcache. Ta je v základě zapnutá od verze 2.26. Sestavení[18] zranitelného programu bylo provedeno na distribuci Fedora 26 (64-bit), gcc verze 7.1.1, staticky slinkováno s glibc 2.25, přiloženy ladící informace. Útok byl otestován v přiloženém virtuálním prostředí.

Ve zranitelném programu jsou alokovány 2 bloky. U prvního dochází k přetečení o jeden byte. Druhý blok musí být větší velikosti, aby nemohl být zařazen po uvolnění do rychlého koše. Dále může útočník nahrát svá data do bufferu o velikosti 32 bytů. Poté dojde k uvolnění druhého bloku. Útočník ještě jednou může zapsat do bufferu na zásobníku. Nakonec je alokován další blok paměti, do kterého útočník zapíše.

Pro prezentaci úspěšného útoku je v programu funkce `challenge()`, kterou nevolá žádná část kódu. Cílem útočníka je její spuštění. Jediné zranitelné místo v programu je přetečení prvního bloku o 1 byte. Pro vygenerování exploitu slouží perlový skript v souboru `exploit.pl`. Z tohoto sou-

[18] $ gcc -g -static houseOfEinherjar.c -o houseOfEinherjar-static-2-25

boru je vytvořený soubor `exploit.in`, sloužící pro zadání vstupu pro program (`$ ./houseOfEinherjat-static-2-25 < exploit.in`).

První programem načítaný vstup je uložen do bufferu na zásobníku. Zde je zapotřebí vytvořit hlavičku falešného chunku. Hodnota `prev_size` není důležitá, `size` je vhodné nastavit na velikost spadající do malého koše, aby se nemusely splňovat kontroly v unlink makru pro velké koše. Důležité je změnit ukazatele `*fd` a `*bk` na adresu chunku, opět z důvodu kontrol v unlink makru.

Druhý načítaný vstup je uložen do `chunkA`. Pro útok je potřeba změnit `chunkB->prev_size` a `PREV_INUSE` bit. Umístění prvního – velikost předchozího chunku – se v případě vhodného zarovnání (platí pro tento program) překrývá s posledními osmi byty (64bitová architektura) předchozího bloku. To je možné, jelikož tento údaj je použit pouze v případě nastaveného `PREV_INUSE` bitu, tedy žádná data se zde již nenachází. Dle velikosti předchozího chunku je vypočtena jeho adresa a tyto chunky jsou spojeny, což má na starosti následující kód:

```
#define chunk_at_offset(p, s)  ((mchunkptr) (((char *) (p)) +
↪  (s)))
// [...]
static void_int_free (mstate av, mchunkptr p, int have_lock) {
  // [...]
  /* consolidate backward */
  if (!prev_inuse(p)) {
    prevsize = prev_size (p);
    size += prevsize;
    p = chunk_at_offset(p, -((long) prevsize));
    unlink(av, p, bck, fwd);
  }
```

Výpis kódu 4.2: Zranitelné spojení s předchozím chunkem (zdrojový soubor glibc v. 2.25 /malloc/malloc.c)

Velikost předchozího chunku je tedy nutné nastavit na hodnotu adresa chunku B (`&chunkB` − 16 bytů) mínus adresa bufferu. Pro změnu `PREV_INUSE` bitu je nutné provést přetečení o 1 byte.

Po této úpravě je `chunkB` uvolněn a došlo ke spojení s falešným chunkem. Během tohoto procesu byla vedlejším efektem změněna velikost tohoto chunku na číslo větší než `av->system_mem`. To by vyvolalo chybu při alokaci z tohoto bloku paměti, proto je nutné jej změnit na validní hodnotu opětovným zápisem do bufferu na zásobníku.

Nyní je již možné provést alokaci další paměti. Pokud vše proběhlo správně, bude tento požadavek uspokojen z předchozího spojeného bloku, který začíná na zásobníku.

Posledním krokem je přepsání zásobníku adresami funkce `challenge()` zápisem do nově alokovaného bloku. Po skončení funkce `main()` je řízení programu přesměrováno do této funkce.

Tabulka 4.2: Struktura exploitu House of Einherjar

Naplnění `buffer[]` (na zásobníku) – falešný chunk	
buffer+0 (fakechunk->prev_size)	8 bytů výplně
buffer+8 (fakechunk->size)	0x200 (spadající do malého koše)
buffer+16 (fakechunk->fd)	adresa `buffer[]`
buffer+24 (fakechunk->bk)	adresa `buffer[]`
Naplnění `chunkA` (na haldě) – přetečení do `chunkB`	
chunkA+0	80 bytů výplně
chunkA+80 (chunkB->prev_size)	adresa `chunkB` - 16 - adresa `buffer[]`
chunkA+88 (chunkB->size)	0x10 (změna `PREV_INUSE` přetečením)
Znovu naplnění `buffer[]` (na zásobníku) – oprava velikosti	
buffer+0 (fakechunk->prev_size)	0x1000
buffer+8 (fakechunk->size)	0x1000
Naplnění `chunkOnStack` (na zásobníku) – přepsání návratové adresy	
chunkOnStack+0	adresa `challenge()`
chunkOnStack+8	adresa `challenge()`
…	adresa `challenge()`
chunkOnStack+56	adresa `challenge()`

4.2.1 Obrana

Glibc od verze 2.26 zavádí používání tcache. Uvolněný `chunkB` nebude vůbec spojován s jiným chunkem a hned bude vložen do tcache. To se ovšem nestane za předpokladu, že bude tcache zaplněna nebo by byla velikost uvolňovaného chunku větší, než je limit pro tcache. Útok je tedy teoreticky možný (za ztížených podmínek) i nadále.

4.3 House of Force

Útok House of Force je funkční i na moderních distribucích. Pouze byla pro zjednodušení ukázky vypnuta ochrana zásobníku, což teoreticky nemusí být nutné. Zranitelný program `houseOfForce-static-2-27` byl zkompilován[19] a staticky slinkován ze zdrojového kódu `houseOfForce.c` přímo ve virtuálním prostředí – gcc verze 7.3.0, glibc verze 2.27, přiloženy ladící informace.

[19] $ gcc -g -static -fno-stack-protector houseOfForce.c -o houseOfForce-static-2-27

Program nejdříve alokuje paměťový blok, do kterého je nahrán vstup od uživatele (parametr programu číslo 1). Tento blok se nachází těsně před top chunkem a délka vstupu není omezena. Pomocí přetečení je možné přepsat velikost top chunku a to klidně na maximální hodnotu (0xffffffffffffffff).

Poté program alokuje další blok paměti, jehož velikost zadává uživatel (parametr 2). Následně je alokovaný 3. blok, do kterého je nahrán další uživatelský vstup (parametr 3).

Základní myšlenkou House of Force je posunutí ukazatele na top chunk. Jelikož je změněna velikost top chunku na abnormálně velkou hodnotu, žádost o druhý blok paměti je uspokojena z top chunku a následně je posunut ukazatel na top chunk za nově alokovaný blok.

Tedy žádostí o alokaci bloku o vhodné velikosti je posunut ukazatel na top chunk například až na zásobník. Velikost alokace odpovídá požadované adrese, kterou má vrátit volání funkce `malloc()` mínus adresa top chunku. Kvůli zarovnání paměti nemusí být tento posun úplně přesný.

Posledním krokem útoku je alokace 3. bloku a zápis do něj. V ukázce je použito přepsání návratové adresy funkce `main()`, kdy je 3. blok (`chunkOnStack`) vyplněn adresou funkce `challenge()`. Vypsáním textu „House of Force“ je demonstrováno provedení útoku.

Ukázkový vstup pro předvedení útoku je ve skriptu `exploit.pl`. Pro spuštění lze použít příkaz `$ ./houseOfForce-static-2-27 `perl exlopit.pl``.

Tabulka 4.3: Struktura exploitu House of Force

Parametr 1: Naplnění `chunkA` – přepsání velikosti top chunku (přetečení)	
chunkA+0	88 bytů výplně
chunkA+88 (top->size)	0xffffffffffffff (změna velikosti top chunku)

Parametr 2: Velikost alokovaného bloku	
	adresa `chunkOnStack` − adresa top chunku

Parametr 3: Přepsání návratové adresy	
	adresa `challenge()`

4.3.1 Obrana

Chyba House of Force byla opravena v poslední verzi glibc (2.29). Do knihovny přibylo ověření, zda není změněna velikost top chunku na hodnotu větší než `av->system_mem`:

```
if (__glibc_unlikely (size > av->system_mem))
  malloc_printerr ("malloc(): corrupted top size");
```

Výpis kódu 4.3: Oprava House of Force (zdrojový soubor glibc v. 2.29 /malloc/malloc.c, ř. 4114-4115)

Proměnná `av->system_mem` určuje velikost dané arény. Hlavní arénu lze rozšiřovat alokací malých bloků paměti, tedy její velikost může narůst až na velikost paměti, kterou poskytne systém.

Teoreticky by bylo možné velikým počtem žádostí o malý blok zvětšit velikost arény. Na 32bitovém systému lze zvětšit hlavní arénu až na cca 3 GB. Pak by byla splněna tato bezpečnostní kontrola a útok by měl být proveditelný. Je samozřejmě otázkou k diskusi, jak moc je tento scénář reálný. Na 64bitovém systému je však tento scénář nereálný, vzhledem k umístění zásobníku oproti haldě není možné alokovat tolik fyzické paměti.

4.4 House of Spirit

Útok House of Spirit je funkční i na aktuální verzi glibc 2.29. Zranitelný program byl zkompilován[20] a staticky slinkován v připraveném virtuálním prostředí – gcc verze 7.3.0, glic verze 2.27, přiloženy ladící informace. Pro zjednodušení útoku byla vypnuta ochrana zásobníku a ASLR.

Zranitelný program vytváří pole `buffer[]` na zásobníku a alokuje blok paměti na haldě. Adresa alokovaného bloku je uložena do ukazatele `*chunk`, který je uložen na zásobníku. Následně jsou do pole `buffer[]` načteny první čtyři parametry programu. Zde je programátorská chyba, dojde k přetečení `buffer[]` o jednu položku. Na 64bitové architektuře je přepsáno 8 bytů za polem, kde se ovšem nachází ukazatel na blok paměti alokovaný na haldě.

Takto změněný ukazatel je předán funkci `free()`. Cílem útočníka je zde uložit adresu falešného chunku, který bude po uvolnění uložen do tcache a při další žádosti o alokaci paměti bude funkcí `malloc()` předán ukazatel na tento falešný blok.

V ukázce je falešný chunk vytvořen na zásobníku. Pro vytvoření tohoto chunku je potřeba mít pod kontrolou pouze 8 bytů, kde bude uložena velikost falešného chunku. Ta musí být rovna velikosti další žádost plus 16 bytů a musí být v rámci velikostí pro tcache. Poté může být změněn ukazatel na uvolňovaný chunk na adresu uživatelských dat falešného chunku, tedy adresu velikosti plus 8 bytů.

Falešný chunk je uvolněn a je uložen do tcache. Další požadavek na alokaci odpovídající velikosti bude uspokojen tímto chunkem. Po alokaci je do nového bloku nahrán uživatelský vstup (parametr 5). Ten přepíše návratovou adresu funkce `main()`.

4.4.1 Obrana

Tento způsob útoku nebyl nikdy opraven. Od zavedení tcache ve verzi 2.26 je dokonce ještě jednodušší na provedení. Falešný chunk nemusí spadat do

[20] `$ gcc -g -static -fno-stack-protector houseOfSpirit.c -o houseOfSpirit-static-2-27`

Tabulka 4.4: Struktura exploitu House of Spirit

Parametr 1, 2, 3, 4: naplnění pole `buffer[]`	
buffer+0	výplň
buffer+8	velikost falešného chunku = budoucí žádost o alokaci + 16)
buffer+16	výplň
buffer+24	přepsání ukazatele na falešný chunk = adresa buffer + 16)

Parametr 5: Přepsání návratové adresy	
	adresa `challenge()`

velikosti rychlých košů, ale může mít velikost až 1 032 nebo 516 bytů (existuje 64 různých velikostí tcache odstupňovaných po 16 bytech na 64bitové architektuře, respektive po 8 bytech na 32bitové architektuře).

Závěr

Tato bakalářská práce se zabývala problematikou přetečení bufferu. V první kapitole byly řešeny možné příčiny a následky přetečení a bylo srovnáno přetečení bufferu na haldě a zásobníku.

Druhá kapitola představuje strukturu haldy knihovny glibc. Tento krok je nezbytný pro vysvětlení útoků v následujících kapitolách. Čtenář je seznámen se základními strukturami haldy, jako je aréna, halda, chunk či koš.

Třetí kapitola naplňuje jeden z cílů práce a to sestavit rešerši známých útoků na haldu knihovny glibc. U každého z útoků je popsána jeho myšlenka včetně ukázek zranitelných částí kódu knihovny glibc. Dále, pokud byla chyba opravena, je informace o způsobu opravy.

Poslední kapitola popisuje vlastní implementaci vybraných útoků. V rámci této bakalářské práce bylo vytvořeno virtuální prostředí, které obsahuje pro vybrané techniky zdrojový kód zranitelného programu, přeložený zranitelný program a skript, který vygeneruje zranitelný vstup. Byly implementovány 4 útoky – tradiční unlink makro, House of Einherjar, House of Force a House of Spirit.

Aktuální verze knihovny glibc je stále zranitelná technikou House of Spirit. House of Force bylo opraveno teprve v nejnovější verzi. Programátor by se měl vyvarovat chybné práci s pamětí. V momentě, kdy dojde k možnosti přetečení, může být zneužití knihovny glibc k provedení útoku reálnou variantou.

Bibliografie

1. ALEPH ONE. Smashing the stack for fun and profit. *Phrack magazine* [online]. 1996, roč. 7, č. 49, s. 14 [cit. 2019-03-04]. Dostupné z: `http://phrack.org/issues/49/14.html`.
2. COWAN, Crispin; WAGLE, Perry; BEATTIE, Steve; WALPOLE, Jonathan. Buffer overflows: attacks and defenses for the vulnerability of the decade. In: *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*. 2000, sv. 2, s. 119–129. ISBN 0-7695-0490-6. Dostupné z DOI: `10.1109/DISCEX.2000.821514`.
3. EISENBERG, Ted; GRIES, David; HARTMANIS, Juris; HOLCOMB, Don; LYNN, M. Stuart; SANTORO, Thomas. The Cornell commission: on Morris and the worm. *Communications of the ACM*. 1989, roč. 32, č. 6, s. 706–709. ISSN 0001-0782. Dostupné z DOI: `10.1145/63526.63530`.
4. *strcpy(3) Linux Programmer's Manual*. 5.00. vyd. 2019.
5. *gets(3) Linux Programmer's Manual*. 5.00. vyd. 2017.
6. SCUT; TEAM TESO. Exploiting format string vulnerabilities [online]. 2001 [cit. 2019-03-07]. Dostupné z: `https://www.win.tue.nl/~aeb/linux/hh/formats-teso.html`.
7. *printf(3) Linux Programmer's Manual*. 5.00. vyd. 2019.
8. FOSTER, James C. et al. *Hacking - Buffer Overflow*. 1. vyd. Praha: Grada, 2007. ISBN 9788024714806.
9. KIRCH, Olaf. The poisoned NUL byte. *Bugtraq Mailing List* [online]. 1998 [cit. 2019-03-08]. Dostupné z: `https://seclists.org/bugtraq/1998/Oct/109`.

10. LU, H. J.; MATZ, Michael; GIRKAR, Milind; HUBIČKA, Jan; JAEGER, Andreas; MITCHELL, Mark. *System V Application Binary Interface: AMD64 Architecture Processor Supplement* [online]. 1.0. vyd. 2018 [cit. 2019-03-11]. Dostupné z: `https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf`.

11. The GNU C Library (glibc). *GNU* [online] [cit. 2019-03-14]. Dostupné z: `https://www.gnu.org/software/libc/`.

12. glibc wiki: MallocInternals. *GNU* [online]. Verze 2018-03-12 [cit. 2019-03-15]. Dostupné z: `https://sourceware.org/glibc/wiki/MallocInternals`.

13. *The GNU C Library Reference Manual: Memory Allocation Tunables* [online]. Verze 2.29 [cit. 2019-03-15]. Dostupné z: `https://www.gnu.org/software/libc/manual/html_node/Memory-Allocation-Tunables.html#index-glibc_002emalloc_002earena_005fmax`.

14. SOLAR DESIGNER. *JPEG COM marker processing vulnerability in Netscape browsers* [online]. 2000 [cit. 2019-03-18]. Dostupné z: `https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability`.

15. ANONYMOUS. Once upon a free(). *Phrack magazine* [online]. 2001, roč. 11, č. 57, s. 9 [cit. 2019-03-18]. Dostupné z: `http://phrack.org/issues/57/9.html`.

16. GNU. *Zdrojové kódy knihovny glibc verze 2.2.5* [online] [cit. 2019-03-20]. Dostupné z: `https://ftp.gnu.org/gnu/glibc/glibc-2.2.5.tar.gz`.

17. *malloc(3) Linux Programmer's Manual*. 5.00. vyd. 2019.

18. GOLLMANN, Dieter. *Computer Security*. 3. vyd. Wiley, 2011. ISBN 978-0-470-74115-3.

19. *CWE-416: Use After Free*. Dostupné také z: `https://cwe.mitre.org/data/definitions/416.html`.

20. PHANTASMAL PHANTASMAGORIA. The Malloc Maleficarum. *Bugtraq mailinglist* [online]. 2005 [cit. 2019-03-20]. Dostupné z: `https://seclists.org/bugtraq/2005/Oct/118`.

21. BLACKNGEL. Malloc Des-Maleficarum. *Phrack magazine* [online]. 2009, roč. 13, č. 66, s. 10 [cit. 2019-03-21]. Dostupné z: `http://phrack.org/issues/66/10.html`.

22. BLACKNGEL. The House Of Lore: Reloaded: ptmalloc v2 & v3: Analysis & Corruption. *Phrack magazine* [online]. 2010, roč. 14, č. 67, s. 8 [cit. 2019-03-30]. Dostupné z: `http://www.phrack.org/issues/67/8.html`.

23. thread local caching in glibc malloc. *Online tukan sanctuary* [online] [cit. 2019-04-06]. Dostupné z: `http://tukan.farm/2017/07/08/tcache/`.

24. MATSUKUMA, Hiroki. *House of Einherjar: Yet Another Heap Exploitation Technique on GLIBC* [online]. 2016 [cit. 2019-04-08]. Dostupné z: `https://www.slideshare.net/codeblue_jp/cb16-matsukuma-en-68459606`.

Dodatek A

Seznam použitých zkratek

ABI Application Binary Interface

ASLR Address Space Layout Randomization

CVE Common Vulnerabilities and Exposures

DEP Data Execution Prevention

glibc GNU C Library

LIFO Last In First Out

LTS Long Term Support

NX Non eXecute

POSIX Portable Operating System Interface

tcache Thread local cache

Dodatek B

Obsah přiloženého DVD

- `readme.txt` ..... stručný popis obsahu DVD
- `impl` ..... adresář s implementací
  - `virt` ..... adresář s virtuálním prostředím
  - `attacks` ..... adresář s implementovanými útoky
- `text` ..... text práce
  - `BP_Bambuch_Michal_2019.pdf` ..... text práce ve formátu PDF
  - `src` ..... zdrojová forma práce ve formátu LaTeX