Ta część omawia koncepcje, które musisz zrozumieć, aby nadać sens reszcie tekstu. Podobnie jak niektóre lektury wymagane na studiach, omówiony tutaj materiał jest wprowadzający i, miejmy nadzieję, już ci znany. Ta część w żadnym wypadku nie jest próbą omówienia wszystkiego, co musisz wiedzieć; powinien raczej służyć jako punkt wyjścia do innych Części. Powinieneś przeczytać to jako przypomnienie. Jeśli znajdziesz pojęcia, które są dla ciebie obce, sugerujemy, abyś oznaczył je jako obszary, w których musisz przeprowadzić więcej badań. Poświęć trochę czasu na zapoznanie się z tymi koncepcjami, zanim przejdziesz dalej.

Podstawowe koncepcje

Aby zrozumieć treść , potrzebujesz dobrze rozwiniętej znajomości języków komputerowych, systemów operacyjnych i architektur. Jeśli nie rozumiesz, jak coś działa, trudno jest wykryć, że działa nieprawidłowo. Dotyczy to komputerów, a także wykrywania i wykorzystywania luk w zabezpieczeniach. Zanim zaczniesz rozumieć pojęcia, musisz znać język. Będziesz musiał znać kilka definicji lub terminów, które są częścią języka badaczy bezpieczeństwa, aby móc lepiej zastosować koncepcje zawarte w tym tekście:

Luka (rzecz.): Luka w zabezpieczeniach systemu, która może prowadzić do tego, że atakujący wykorzysta system w sposób inny niż zamierzony przez projektanta. Może to obejmować wpływ na dostępność systemu, podniesienie uprawnień dostępu do niezamierzonego poziomu, pełną kontrolę systemu przez nieuprawnioną stronę i wiele innych możliwości. Znany również jako luka w zabezpieczeniach lub błąd w zabezpieczeniach.

Exploit (czas.): Wykorzystanie luki w zabezpieczeniach, aby system docelowy reagował w sposób inny niż zamierzony przez projektanta.

Exploit (rzecz.): Narzędzie, zestaw instrukcji lub kod używany do wykorzystania luki w zabezpieczeniach. Znany również jako Proof of Concept (POC).

0day (rzecz.): Exploit wykorzystujący lukę, która nie została ujawniona publicznie. Czasami używany w odniesieniu do samej luki.

Fuzzer (rzecz.): Narzędzie lub aplikacja, która próbuje wszystkich lub szerokiego zakresu nieoczekiwanych wartości wejściowych do systemu. Celem fuzzera jest ustalenie, czy w systemie istnieje błąd, który można później wykorzystać bez konieczności pełnego poznania wewnętrznego funkcjonowania systemu docelowego.

INSTRUKCJE I DANE

Nowoczesny komputer nie rozróżnia instrukcji i danych. Jeśli procesor może otrzymać instrukcje, kiedy powinien widzieć dane, z radością wykona przekazane instrukcje. Ta cecha umożliwia eksploatację systemu. Ten tekst uczy, jak wstawiać instrukcje, gdy projektant systemu oczekiwał danych. Wykorzystasz również koncepcję przepełnienia, aby nadpisać instrukcje projektanta własnymi. Celem jest przejęcie kontroli nad wykonaniem.

Zarządzanie pamięcią

Aby skorzystać z tekstu, musisz zrozumieć nowoczesne zarządzanie pamięcią, szczególnie dla 32-bitowej architektury Intel (IA32). Linux na IA32 jest omówiony wyłącznie w pierwszej części i użyty w częściach wprowadzających. Musisz zrozumieć, jak zarządza się pamięcią, ponieważ większość luk w zabezpieczeniach opisanych wynika z nadpisywania lub przepełniania jednej części pamięci drugą. Gdy program jest wykonywany, jest on zorganizowany w sposób zorganizowany – różne elementy programu są mapowane do pamięci. Najpierw system operacyjny tworzy przestrzeń adresową, w

której będzie działał program. Ta przestrzeń adresowa zawiera rzeczywiste instrukcje programu, jak również wszelkie wymagane dane. Następnie informacje są ładowane z pliku wykonywalnego programu do nowo utworzonej przestrzeni adresowej. Istnieją trzy typy segmentów: .text, .bss i .data. Segment .text jest mapowany jako tylko do odczytu, podczas gdy .data i .bss są zapisywalne. Segmenty .bss i .data są zarezerwowane dla zmiennych globalnych. Segment .data zawiera statyczne zainicjowane dane, a segment .bss zawiera niezainicjowane dane. Ostatni segment, .text, zawiera instrukcje programu. Na koniec inicjowany jest stos i sterta. Stos jest strukturą danych, a dokładniej strukturą danych Last In First Out (LIFO), co oznacza, że najnowsze dane umieszczone lub wypchnięte na stos są kolejnym elementem do usunięcia lub usunięcia ze stosu. Struktura danych LIFO jest idealna do przechowywania informacji przejściowych lub informacji, które nie muszą być przechowywane przez dłuższy czas. Stos przechowuje zmienne lokalne, informacje dotyczące wywołań funkcji i inne informacje używane do czyszczenia stosu po wywołaniu funkcji lub procedury. Inną ważną cechą stosu jest to, że rośnie on w dół przestrzeni adresowej: im więcej danych jest dodawanych do stosu, są one dodawane przy coraz niższych wartościach adresu. Sterta to kolejna struktura danych używana do przechowywania informacji o programie, a dokładniej zmiennych dynamicznych. Sterta jest (w przybliżeniu) strukturą danych First In First Out (FIFO). Dane są umieszczane i usuwane ze sterty w miarę jej budowania. Sterta powiększa przestrzeń adresową: Gdy dane są dodawane do sterty, są one dodawane z coraz większą wartością adresu, jak pokazano na poniższym diagramie przestrzeni pamięci.

↑ Lower addresses (0x08000000)

Shared libraries

.text

.bss

Heap (grows ↓)

Stack (grows ↑)

env pointer

Argc

↓ Higher addresses (0xbfffffff)

Zarządzanie pamięcią przedstawione w tej części musi być rozumiane na znacznie głębszym, bardziej szczegółowym poziomie, aby w pełni zrozumieć, a co ważniejsze, zastosować to, co jest zawarte tu.

Asemblacja

Znajomość języka asemblera specyficznego dla IA32 jest wymagana do zrozumienia dużej części tego tekstu. Duża część procesu wykrywania błędów obejmuje interpretację i zrozumienie asemblera, a znaczna część skupia się na asemblerze z 32-bitowym procesorem Intel. Wykorzystywanie luk w zabezpieczeniach wymaga dobrej znajomości języka asemblera, ponieważ większość exploitów będzie wymagać od ciebie napisania (lub zmodyfikowania istniejącego) kodu w asemblerze. Ponieważ systemy inne niż IA32 są ważne, ale mogą być nieco trudniejsze do wykorzystania, tekst obejmuje również wykrywanie i wykorzystywanie błędów w innych rodzinach procesorów. Jeśli planujesz prowadzić badania bezpieczeństwa na innych platformach, ważne jest, abyś dobrze rozumiał zestaw specyficzny dla wybranej architektury. Jeśli nie jesteś dobrze zorientowany lub nie masz doświadczenia z asemblerem, najpierw musisz nauczyć się systemów liczbowych (w szczególności szesnastkowych),

rozmiarów danych i reprezentacji znaków liczbowych. Te koncepcje inżynierii komputerowej można znaleźć w większości książek o architekturze komputerowej na poziomie uczelni.

Rejestry

Zrozumienie, w jaki sposób rejestry działają na procesorze IA32 i w jaki sposób są manipulowane przez asembler, jest niezbędne do opracowywania i wykorzystywania luk. Rejestry mogą być dostępne, odczytywane i zmieniane za pomocą asemblera. Rejestry to pamięć, zwykle podłączona bezpośrednio do obwodów ze względu na wydajność. Odpowiadają za manipulacje, które umożliwiają funkcjonowanie nowoczesnych komputerów i można nimi manipulować za pomocą instrukcji montażu. Z wysokiego poziomu rejestry można podzielić na cztery kategorie:

* Ogólnego przeznaczenia

* Segment

* Sterowanie

* Inne

Rejestry ogólnego przeznaczenia służą do wykonywania szeregu typowych operacji matematycznych. Obejmują one rejestry takie jak EAX, EBX i ECX dla IA32 i mogą być używane do przechowywania danych i adresów, adresów przesunięcia, wykonywania funkcji zliczania i wielu innych rzeczy. Rejestr ogólnego przeznaczenia, o którym należy pamiętać, to rejestr rozszerzonego wskaźnika stosu (ESP) lub po prostu wskaźnik stosu. ESP wskazuje adres pamięci, pod którym odbędzie się następna operacja na stosie. Aby zrozumieć przepełnienia stosu, powinieneś dokładnie zrozumieć, w jaki sposób ESP jest używany z typowymi instrukcjami asemblera i jaki ma wpływ na dane przechowywane na stosie. Kolejną klasą rejestru zainteresowań jest rejestr segmentowy. W przeciwieństwie do innych rejestrów w procesorze IA32, rejestry segmentowe są 16 bitowe (inne rejestry mają rozmiar 32 bity). Rejestry segmentowe, takie jak CS, DS i SS, służą do śledzenia segmentów i zapewnienia kompatybilności wstecznej z aplikacjami 16-bitowymi. Rejestry kontrolne służą do sterowania funkcją procesora. Najważniejszym z tych rejestrów dla IA32 jest rozszerzony wskaźnik instrukcji (EIP) lub po prostu wskaźnik instrukcji. EIP zawiera adres następnej instrukcji maszynowej do wykonania. Oczywiście, jeśli chcesz kontrolować ścieżkę wykonania programu, o czym nawiasem mówiąc jest ten tekst, ważne jest, aby mieć możliwość dostępu i zmiany wartości przechowywanej w rejestrze EIP. Rejestry z drugiej kategorii to po prostu obce rejestry, które nie pasują dokładnie do pierwszych trzech kategorii. Jednym z tych rejestrów jest rejestr Extended Flags (EFLAGS), który składa się z wielu rejestrów jednobitowych służących do przechowywania wyników różnych testów wykonywanych przez procesor. Gdy już dobrze zrozumiesz rejestry, możesz przejść dalej , do samego programowania w asemblerze.

Rozpoznawanie konstrukcji kodu C i C++ w asemblerze

Rodzina języków programowania C (C, C++, C#) jest jednym z najczęściej używanych, jeśli nie najczęściej używanym gatunkiem języków programowania. C jest zdecydowanie najpopularniejszym językiem dla aplikacji serwerowych Windows i Unix, które są dobrym celem dla rozwijania podatności. Z tych powodów gruntowne zrozumienie języka C ma kluczowe znaczenie. Wraz z szerokim zrozumieniem języka C powinieneś być w stanie zrozumieć, jak skompilowany kod C przekłada się na asembler. Zrozumienie, w jaki sposób zmienne, wskaźniki, funkcje i alokacja pamięci w języku C są reprezentowane przez asembler, znacznie ułatwi zrozumienie zawartości tej książki. Weźmy kilka typowych konstrukcji kodu C i C++ i zobaczmy, jak wyglądają w asemblerze. Jeśli dobrze znasz te przykłady, powinieneś być gotowy, aby przejść dalej.

Spójrzmy na deklarację liczby całkowitej w C++, a następnie użycie tej samej liczby całkowitej do liczenia:

```
int number;

… more code …

number++;
```

Można to przełożyć na, w asemblerze:

```
number dw 0

… more code …

mov eax,number

inc eax

mov number,eax
```

Używamy instrukcji Define Word (DW) do zdefiniowania wartości dla naszej liczby całkowitej, liczby. Następnie wstawiamy wartość do rejestru EAX, zwiększamy wartość w rejestrze EAX o jeden, a następnie przenosimy tę wartość z powrotem do liczby całkowitej. Spójrz na prostą instrukcję if w C++:

```
int number;

if (number < 0)

{

… more code …

}
```

Teraz spójrz na tę samą instrukcję if w asemblerze:

```
number dw 0

mov eax,number

or eax,eax

jge label

< no >

label :< yes >
```

To, co robimy tutaj, to ponowne zdefiniowanie wartości liczby za pomocą instrukcji DW. Następnie przenosimy wartość zapisaną w numerze do EAX, a następnie przeskakujemy do etykiety, jeśli liczba jest większa lub równa zero za pomocą opcji Skok, jeśli jest większa lub równa (JGE). Oto kolejny przykład, używając tablicy:

```
int array[4];

… more code…

array[2]=9;
```

Tutaj zadeklarowaliśmy tablicę, tablicę i ustawiliśmy element tablicy równy 9. W asemblerze mamy:

```
array dw 0,0,0,0

… more code …

mov ebx,2

mov array[ebx],9
```

W tym przykładzie deklarujemy tablicę, a następnie używamy rejestru EBX do przenoszenia wartości do tablicy. Na koniec spójrzmy na bardziej skomplikowany przykład. Kod pokazuje, jak wygląda prosta funkcja C w asemblerze. Jeśli z łatwością zrozumiesz ten przykład, prawdopodobnie jesteś gotowy, aby przejść do następnej części

```
int triangle (int width, in height){

int array[5] = {0,1,2,3,4};

int area;

area = width * height/2;

return (area);

}
```

Oto ta sama funkcja, ale w formie zdemontowanej. Poniższe dane są wysyłane z debugera gdb. gdb to debugger projektu GNU. Zobacz, czy możesz dopasować asembler do kodu C:

```
0x8048430 < triangle >: push %ebp

0x8048431 < triangle+1 >: mov %esp, %ebp

0x8048433 < triangle+3 >: push %edi

0x8048434 < triangle+4 >: push %esi

0x8048435 < triangle+5 >: sub $0x30,%esp

0x8048438 < triangle+8 >: lea 0xffffffd8(%ebp), %edi

0x804843b < triangle+11 >: mov $0x8049508,%esi

0x8048440 < triangle+16 >: cld

0x8048441 < triangle+17 >: mov $0x30,%esp

0x8048446 < triangle+22 >: repz movsl %ds:( %esi), %es:( %edi)

0x8048448 < triangle+24 >: mov 0x8(%ebp),%eax

0x804844b < triangle+27 >: mov %eax,%edx

0x804844d < triangle+29 >: imul 0xc(%ebp),%edx

0x8048451 < triangle+33 >: mov %edx,%eax

0x8048453 < triangle+35 >: sar $0x1f,%eax
```

0x8048456 < triangle+38 >: shr $0x1f,%eax

0x8048459 < triangle+41 >: lea (%eax, %edx, 1), %eax

0x804845c < triangle+44 >: sar %eax

0x804845e < triangle+46 >: mov %eax,0xfffffffd4(%ebp)

0x8048461 < triangle+49 >: mov 0xfffffffd4(%ebp),%eax

0x8048464 < triangle+52 >: mov %eax,%eax

0x8048466 < triangle+54 >: add $0x30,%esp

0x8048469 < triangle+57 >: pop %esi

0x804846a < triangle+58 >: pop %edi

0x804846b < triangle+59 > pop %ebp

0x804846c < triangle+60 >: ret

Najważniejszą rzeczą, jaką robi funkcja, jest mnożenie dwóch liczb, więc zwróć uwagę na instrukcję imul pośrodku. Zwróć też uwagę na kilka pierwszych instrukcji - zapisywanie EBP i odejmowanie od ESP. Odejmowanie robi miejsce na stosie dla zmiennych lokalnych funkcji. Warto również zauważyć, że funkcja zwraca swój wynik w rejestrze EAX.

Wniosek

Wprowadziliśmy kilka podstawowych pojęć, które musisz znać, aby zrozumieć resztę tekstu. Powinieneś poświęcić trochę czasu na przejrzenie pojęć tu przedstawionych. Jeśli stwierdzisz, że nie masz wystarczającej ekspozycji na język asemblerowy i C lub C++, być może będziesz musiał zrobić pewne przygotowanie w tle, aby uzyskać pełną wartość z następnych części.