Atakowanie natywnych aplikacji skompilowanych

Skompilowane oprogramowanie, które działa w natywnym środowisku wykonawczym, było w przeszłości nękane lukami w zabezpieczeniach, takimi jak przepełnienie bufora i błędy w formatowaniu ciągów znaków. Większość aplikacji internetowych jest napisanych przy użyciu języków i platform, które działają w zarządzanym środowisku wykonawczym, w którym nie występują te klasyczne luki w zabezpieczeniach. Jedną z najbardziej znaczących zalet języków, takich jak C# i Java, jest to, że programiści nie muszą się martwić o zarządzanie buforami i problemy z arytmetyką wskaźników, które miały wpływ na oprogramowanie tworzone w językach rodzimych, takich jak C i C++, i które dały początek do większości krytycznych błędów znalezionych w tym oprogramowaniu. Niemniej jednak czasami możesz napotkać aplikacje internetowe napisane w kodzie natywnym. Ponadto wiele aplikacji napisanych głównie przy użyciu kodu zarządzanego zawiera fragmenty kodu natywnego lub wywołuje komponenty zewnętrzne, które działają w kontekście niezarządzanym. O ile nie masz pewności, że Twoja aplikacja docelowa nie zawiera żadnego kodu natywnego, warto wykonać kilka podstawowych testów zaprojektowanych w celu wykrycia wszelkich klasycznych luk, które mogą istnieć. Aplikacje internetowe działające na urządzeniach sprzętowych, takich jak drukarki i przełączniki, często zawierają pewien kod natywny. Inne prawdopodobne cele obejmują każdą stronę lub skrypt, którego nazwa zawiera możliwe wskaźniki kodu natywnego, takie jak dll lub exe, oraz wszelkie funkcje, o których wiadomo, że wywołują starsze komponenty zewnętrzne, takie jak mechanizmy rejestrowania. Jeśli uważasz, że aplikacja, którą atakujesz, zawiera znaczne ilości kodu natywnego, pożądane może być przetestowanie każdego fragmentu danych dostarczonych przez użytkownika przetwarzanych przez aplikację, w tym nazw i wartości każdego parametru, pliku cookie, nagłówka żądania i innych dane. W tym rozdziale omówiono trzy główne kategorie klasycznych luk w zabezpieczeniach oprogramowania: przepełnienie bufora, luki w zabezpieczeniach liczb całkowitych i błędy w formatowaniu ciągów znaków. W każdym przypadku opiszemy kilka typowych luk w zabezpieczeniach, a następnie przedstawimy praktyczne kroki, które można podjąć, szukając tych błędów w aplikacji internetowej. Ten temat jest ogromny i wykracza daleko poza zakres książki o hakowaniu aplikacji internetowych.

UWAGA: Zdalne sondowanie luk w zabezpieczeniach opisanych w tym rozdziale niesie ze sobą wysokie ryzyko odmowy usługi dla aplikacji. W przeciwieństwie do luk, takich jak słabe uwierzytelnianie i przemierzanie ścieżek, samo wykrycie klasycznych luk w oprogramowaniu może spowodować nieobsługiwane wyjątki w docelowej aplikacji, co może spowodować, że przestanie ona działać. Jeśli zamierzasz sondować działającą aplikację pod kątem tych błędów, przed rozpoczęciem musisz upewnić się, że właściciel aplikacji akceptuje ryzyko związane z testowaniem.

Luki w zabezpieczeniach związane z przepełnieniem bufora

Luki w zabezpieczeniach związane z przepełnieniem bufora występują, gdy aplikacja kopiuje dane kontrolowane przez użytkownika do bufora pamięci, który nie jest wystarczająco duży, aby je pomieścić. Bufor docelowy jest przepełniony, co powoduje nadpisanie sąsiedniej pamięci danymi użytkownika. W zależności od charakteru luki osoba atakująca może ją wykorzystać do wykonania dowolnego kodu na serwerze lub wykonania innych nieautoryzowanych działań. Luki w zabezpieczeniach związane z przepełnieniem bufora są od lat bardzo rozpowszechnione w oprogramowaniu natywnym i są powszechnie uważane za wroga publicznego numer jeden, którego twórcy takiego oprogramowania muszą unikać.

Przepełnienie stosu

Przepełnienie bufora zwykle występuje, gdy aplikacja używa nieograniczonej operacji kopiowania (takiej jak strcpy w C) w celu skopiowania bufora o zmiennej wielkości do bufora o stałym rozmiarze

bez sprawdzania, czy bufor o stałym rozmiarze jest wystarczająco duży. Na przykład następująca funkcja kopiuje ciąg nazwy użytkownika do bufora o stałym rozmiarze przydzielonego na stosie:

```
bool CheckLogin(char* username, char* password)

{

char _username[32];

strcpy(_username, username);

...
```

Jeśli ciąg nazwy użytkownika zawiera więcej niż 32 znaki, bufor _username jest przepełniony, a atakujący nadpisuje dane w sąsiedniej pamięci. W przypadku przepełnienia bufora opartego na stosie udany exploit zazwyczaj polega na zastąpieniu zapisanego adresu zwrotnego na stosie. Gdy wywoływana jest funkcja CheckLogin, procesor umieszcza na stosie adres instrukcji następującej po wywołaniu. Po zakończeniu CheckLogin procesor zdejmuje ten adres ze stosu i zwraca wykonanie tej instrukcji. W międzyczasie funkcja CheckLogin przydziela bufor _username na stosie tuż obok zapisanego adresu zwrotnego. Jeśli atakujący może przepełnić bufor _username, może nadpisać zapisany adres zwrotny wybraną przez siebie wartością, powodując w ten sposób przeskok procesora do tego adresu i wykonanie dowolnego kodu.

Przepełnienie sterty

Przepełnienia bufora oparte na stercie zasadniczo obejmują ten sam rodzaj niebezpiecznej operacji, co opisano wcześniej, z tym wyjątkiem, że przepełniony bufor docelowy jest przydzielany na stercie, a nie na stosie:

```
bool CheckLogin(char* username, char* password)

{

char* _username = (char*) malloc(32);

strcpy(_username, username);

...
```

W przypadku przepełnienia bufora opartego na stercie, tym, co zwykle sąsiaduje z buforem docelowym, nie jest żaden zapisany adres powrotu, ale inne bloki pamięci sterty, oddzielone strukturami kontrolnymi sterty. Sterta jest zaimplementowana jako podwójnie połączona lista: każdy blok jest poprzedzony w pamięci strukturą kontrolną, która zawiera rozmiar bloku, wskaźnik do poprzedniego bloku na stercie i wskaźnik do następnego bloku na stercie. Kiedy bufor sterty jest przepełniony, struktura kontrolna sąsiedniego bloku sterty jest nadpisywana danymi kontrolowanymi przez użytkownika. Ten typ luki jest trudniejszy do wykorzystania niż przepełnienie stosu, ale powszechnym podejściem jest zapisanie spreparowanych wartości w nadpisanej strukturze kontrolnej sterty, aby spowodować arbitralne nadpisanie krytycznego wskaźnika w przyszłości. Gdy blok sterty, którego struktura kontrolna została nadpisana, zostanie zwolniony z pamięci, menedżer sterty musi zaktualizować połączoną listę bloków sterty. Aby to zrobić, musi zaktualizować wskaźnik łącza wstecznego następnego bloku sterty i zaktualizować wskaźnik łącza przedniego poprzedniego bloku sterty, tak aby te dwa elementy na połączonej liście wskazywały na siebie. W tym celu menedżer sterty używa wartości w nadpisanej strukturze kontrolnej. W szczególności, aby zaktualizować wskaźnik łącza zwrotnego następującego bloku, menedżer sterty dereferuje wskaźnik łącza nadawczego pobrany z

nadpisanej struktury kontrolnej i zapisuje do struktury pod tym adresem wartość wskaźnika łącza zwrotnego pobranego z nadpisanej struktury kontrolnej. Innymi słowy, zapisuje kontrolowaną przez użytkownika wartość do kontrolowanego przez użytkownika adresu. Jeśli atakujący odpowiednio spreparował swoje dane dotyczące przepełnienia, może nadpisać dowolny wskaźnik w pamięci wybraną przez siebie wartością w celu przejęcia kontroli nad ścieżką wykonania, a tym samym wykonania dowolnego kodu. Typowe cele nadpisania dowolnego wskaźnika to wartość wskaźnika funkcji, który później wywoła aplikacja, oraz adres programu obsługi wyjątków, który zostanie wywołany przy następnym wystąpieniu wyjątku.

UWAGA: Nowoczesne kompilatory i systemy operacyjne mają zaimplementowane różne zabezpieczenia w celu ochrony oprogramowania przed błędami programistycznymi, które prowadzą do przepełnienia bufora. Te mechanizmy obronne oznaczają, że obecnie rzeczywiste przepełnienia są generalnie trudniejsze do wykorzystania niż opisane tutaj przykłady. Aby uzyskać więcej informacji na temat tych zabezpieczeń i sposobów ich obejścia

Luki w zabezpieczeniach typu „off-by-one”.

Specyficzny rodzaj luki w zabezpieczeniach związanej z przepełnieniem pojawia się, gdy błąd programistyczny umożliwia atakującemu zapisanie pojedynczego bajtu (lub niewielkiej liczby bajtów) poza końcem przydzielonego bufora. Rozważmy następujący kod, który przydziela bufor na stosie, wykonuje zliczoną operację kopiowania bufora, a następnie kończy łańcuch docelowy znakiem null:

```
bool CheckLogin(char* username, char* password)

{

char _username[32];

int i;

for (i = 0; username[i] && i < 32; i++)

_username[i] = username[i];

_username[i] = 0;

...
```

Kod kopiuje do 32 bajtów, a następnie dodaje terminator zerowy. Stąd, jeśli nazwa użytkownika ma 32 bajty lub więcej, bajt zerowy jest zapisywany poza końcem bufora _username, uszkadzając sąsiednią pamięć. Ten warunek może być wykorzystany. Jeśli sąsiedni element na stosie jest wskaźnikiem zapisanej ramki ramki wywołującej, ustawienie bajtu niższego rzędu na zero może spowodować, że będzie on wskazywał na bufor _username, a tym samym na dane kontrolowane przez atakującego. Kiedy wywołanie funkcji powróci, może to umożliwić osobie atakującej przejęcie kontroli nad przebiegiem wykonywania. Podobny rodzaj luki pojawia się, gdy programiści przeoczą potrzebę, aby bufory ciągów zawierały miejsce na terminator zerowy. Rozważ następującą „poprawkę” oryginalnego przepełnienia sterty:

```
bool CheckLogin(char* username, char* password)

{

char* _username = (char*) malloc(32);

strncpy(_username, username, 32);
```

...

W tym przypadku programista tworzy bufor o stałej wielkości na stercie, a następnie wykonuje zliczoną operację kopiowania bufora, zaprojektowaną w celu zapewnienia, że bufor nie zostanie przepełniony. Jeśli jednak nazwa użytkownika jest dłuższa niż bufor, bufor jest całkowicie wypełniony znakami z nazwy użytkownika, nie pozostawiając miejsca na dołączenie końcowego bajtu zerowego. W związku z tym skopiowana wersja ciągu utraciła terminator o wartości null. Języki takie jak C nie mają oddzielnego zapisu długości łańcucha. Koniec łańcucha jest wskazywany przez bajt zerowy (czyli bajt z kodem znaku ASCII zero). Jeśli łańcuch traci swój terminator zerowy, efektywnie zwiększa swoją długość i kontynuuje aż do następnego bajtu w pamięci, który jest równy zero. Ta niezamierzona konsekwencja może często powodować nietypowe zachowanie i luki w zabezpieczeniach aplikacji. Autorzy natrafili na tego typu lukę w aplikacji internetowej działającej na urządzeniu sprzętowym. Aplikacja zawierała stronę, która przyjmowała dowolne parametry w żądaniu POST i zwracała formularz HTML zawierający nazwy i wartości tych parametrów jako pola ukryte. Na przykład:

POST /formRelay.cgi HTTP/1.0

Content-Length: 3

a=b

HTTP/1.1 200 OK

Date: THU, 01 SEP 2011 14:53:13 GMT

Content-Type: text/html

Content-Length: 278

<html>

<head>

<meta http-equiv=”content-type” content=”text/html;charset=iso-8859-1”>

</head>

<form name=”FORM_RELAY” action=”page.cgi” method=”POST”>

<input type=”hidden” name=”a” value=”b”>

</form>

<body onLoad=”document.FORM_RELAY.submit();”>

</body>

</html>

Z jakiegoś powodu ta strona była używana w całej aplikacji do przetwarzania wszelkiego rodzaju danych wprowadzanych przez użytkowników, z których większość była poufna. Jeśli jednak przesłano 4096 lub więcej bajtów danych, zwrócony formularz zawierał również parametry przesłane przez poprzednie żądanie do strony, nawet jeśli zostały one przesłane przez innego użytkownika. Na przykład:

POST /formRelay.cgi HTTP/1.0

Content-Length: 4096

a=bbbbbbbbbbbbb[lots more b’s]

HTTP/1.1 200 OK

Date: THU, 01 SEP 2011 14:58:31 GMT

Content-Type: text/html

Content-Length: 4598

<html>

<head>

<meta http-equiv=”content-type” content=”text/html;charset=iso-8859-1”>

</head>

<form name=”FORM_RELAY” action=”page.cgi” method=”POST”>

<input type=”hidden” name=”a” value=”bbbbbbbbbbbbb[lots more b’s]”>

<input type=”hidden” name=”strUsername” value=”agriffiths”>

<input type=”hidden” name=”strPassword” value=”aufwiedersehen”>

<input type=“hidden“ name=“Log_in“ value=“Log+In“>

</form>

<body onLoad=“document.FORM_RELAY.submit();“>

</body>

</html>

Po zidentyfikowaniu tej luki możliwe było ciągłe sondowanie podatnej strony za pomocą zbyt długich danych i analizowanie odpowiedzi w celu zarejestrowania każdego fragmentu danych przesłanych na stronę przez innych użytkowników. Obejmuje to dane logowania i inne poufne informacje. Podstawową przyczyną luki było to, że dane dostarczone przez użytkownika były przechowywane jako łańcuchy zakończone znakiem null w 4096-bajtowych blokach pamięci. Dane zostały skopiowane w sprawdzonej operacji, więc nie było możliwe proste przepełnienie. Jeśli jednak przesłano zbyt długie dane wejściowe, operacja kopiowania powodowała utratę terminatora zerowego, więc ciąg płynął do następnych danych w pamięci. W związku z tym, gdy aplikacja analizowała parametry żądania, była kontynuowana aż do następnego bajtu zerowego, w związku z czym zawierała parametry dostarczone przez innego użytkownika

Wykrywanie luk w zabezpieczeniach związanych z przepełnieniem buforu

Podstawową metodologią wykrywania luk w zabezpieczeniach związanych z przepełnieniem bufora jest wysyłanie długich ciągów danych do zidentyfikowanego celu i monitorowanie pod kątem nieprawidłowych wyników. W niektórych przypadkach istnieją subtelne luki w zabezpieczeniach, które można wykryć tylko przez wysłanie zbyt długiego łańcucha o określonej długości lub w niewielkim zakresie długości. Jednak w większości przypadków luki w zabezpieczeniach można wykryć po prostu wysyłając ciąg znaków dłuższy niż oczekuje aplikacja. Programiści zwykle tworzą bufory o stałym

rozmiarze, używając okrągłych liczb dziesiętnych lub szesnastkowych, takich jak 32, 100, 1024, 4096 i tak dalej. Prostym podejściem do wykrywania „nisko wiszących owoców” w aplikacji jest wysyłanie długich ciągów znaków w miarę identyfikowania każdego elementu danych docelowych oraz monitorowanie odpowiedzi serwera pod kątem anomalii

KROKI HACKOWANIA

1. Dla każdego docelowego elementu danych prześlij zakres długich ciągów o długości nieco większej niż typowe rozmiary buforów. Na przykład:

1100

4200

33000

2. Celuj w jeden element danych na raz, aby zmaksymalizować pokrycie ścieżek kodu w aplikacji.

3. Możesz użyć źródła ładunku bloków znaków w Burp Intruder do automatycznego generowania ładunków o różnych rozmiarach.

4. Monitoruj odpowiedzi aplikacji, aby zidentyfikować wszelkie anomalie. Niekontrolowane przepełnienie prawie na pewno spowoduje wyjątek w aplikacji. Wykrywanie, kiedy to miało miejsce w procesie zdalnym, jest trudne, ale oto kilka nietypowych zdarzeń, których należy szukać:

* Kod stanu HTTP 500 lub komunikat o błędzie, gdy inne źle sformułowane (ale nie za długie) dane wejściowe nie mają takiego samego efektu

* Komunikat informacyjny, wskazujący, że w niektórych wystąpiła awaria

natywny komponent kodu

* Z serwera otrzymano częściową lub zniekształconą odpowiedź

* Połączenie TCP z serwerem zostaje nagle przerwane bez odpowiedzi

* Cała aplikacja internetowa przestaje odpowiadać

5. Należy zauważyć, że gdy zostanie wyzwolone przepełnienie sterty, może to spowodować awarię w pewnym momencie w przyszłości, a nie natychmiast. Może być konieczne przeprowadzenie eksperymentu w celu zidentyfikowania jednego lub kilku przypadków testowych powodujących uszkodzenie sterty.

6. Luka typu off-by-one może nie spowodować awarii, ale może skutkować nietypowym zachowaniem, takim jak nieoczekiwane zwrócenie danych przez aplikację.

W niektórych przypadkach przypadki testowe mogą zostać zablokowane przez kontrole poprawności danych wejściowych zaimplementowane w samej aplikacji lub przez inne komponenty, takie jak serwer WWW. Dzieje się tak często, gdy w ciągu zapytania adresu URL przesyłane są zbyt długie dane i może być wskazywane przez ogólny komunikat, taki jak „Zbyt długi adres URL” w odpowiedzi na każdy przypadek testowy. W takiej sytuacji należy poeksperymentować, aby określić maksymalną dozwoloną długość adresu URL (która często wynosi około 2000 znaków) i dostosować rozmiary buforów, aby przypadki testowe spełniały to wymaganie. Za ogólnym filtrowaniem długości mogą nadal występować przepełnienia, które mogą być wyzwalane przez łańcuchy wystarczająco krótkie, aby ominąć to filtrowanie. W innych przypadkach filtry mogą ograniczać typ danych lub zakres znaków, które można przesłać w ramach określonego parametru. Na przykład aplikacja może sprawdzać, czy przesłana

nazwa użytkownika zawiera tylko znaki alfanumeryczne przed przekazaniem jej do funkcji zawierającej przepełnienie. Aby zmaksymalizować skuteczność testowania, należy starać się upewnić, że każdy przypadek testowy zawiera tylko znaki dozwolone w odpowiednim parametrze. Jedną ze skutecznych technik osiągnięcia tego celu jest przechwycenie normalnego żądania zawierającego dane, które aplikacja akceptuje, i rozszerzenie każdego docelowego parametru z kolei przy użyciu tych samych znaków, które już zawiera, w celu utworzenia długiego ciągu, który prawdopodobnie przejdzie przez wszelkie filtry oparte na treści . Nawet jeśli masz pewność, że występuje stan przepełnienia bufora, zdalne wykorzystanie go do wykonania dowolnego kodu jest niezwykle trudne. Peter Winter-Smith z NGSSoftware przeprowadził kilka interesujących badań dotyczących możliwości wykorzystania ślepego przepełnienia bufora.

Luki w zabezpieczeniach liczb całkowitych

Luki w zabezpieczeniach związane z liczbami całkowitymi zwykle pojawiają się, gdy aplikacja wykonuje pewne działania arytmetyczne na wartości długości przed wykonaniem operacji na buforze, ale nie bierze pod uwagę pewnych cech sposobu, w jaki kompilatory i procesory obsługują liczby całkowite. Warto zwrócić uwagę na dwa rodzaje błędów całkowitoliczbowych: przepełnienia i błędy podpisu.

Przepełnienia całkowitoliczbowe

Występują one, gdy operacja na wartości całkowitej powoduje jej wzrost powyżej maksymalnej możliwej wartości lub spadek poniżej minimalnej możliwej wartości. Kiedy to nastąpi, liczba zawija się, więc bardzo duża liczba staje się bardzo mała lub odwrotnie. Rozważ następującą „naprawę" opisanego wcześniej przepełnienia sterty:

```
bool CheckLogin(char* username, char* password)

{

unsigned short len = strlen(username) + 1;

char* _username = (char*) malloc(len);

strcpy(_username, username);

...
```

Tutaj aplikacja mierzy długość nazwy użytkownika przesłanej przez użytkownika, dodaje 1, aby pomieścić końcową wartość null, przydziela bufor o wynikowym rozmiarze, a następnie kopiuje do niego nazwę użytkownika. W przypadku danych wejściowych o normalnym rozmiarze ten kod zachowuje się zgodnie z przeznaczeniem. Jeśli jednak użytkownik poda nazwę użytkownika złożoną z 65 535 znaków, nastąpi przepełnienie liczby całkowitej. Krótka liczba całkowita zawiera 16 bitów, co jest wystarczające aby jego wartość mieściła się w przedziale od 0 do 65 535. Kiedy przesyłany jest ciąg znaków o długości 65 535, program dodaje do tego 1, a wartość zawija się do 0. Alokowany jest bufor o zerowej długości i kopiowana jest do niego długa nazwa użytkownika, co powoduje przepełnienie sterty. Atakujący skutecznie udaremnił próbę programisty upewnienia się, że bufor docelowy jest wystarczająco duży. Błędy znaku Występują, gdy aplikacja używa zarówno liczb całkowitych ze znakiem, jak i bez znaku do pomiaru długości buforów i myli je w pewnym momencie. Albo aplikacja dokonuje bezpośredniego porównania wartości ze znakiem i bez znaku, albo przekazuje wartość ze znakiem jako parametr do funkcji, która przyjmuje wartość bez znaku. W obu przypadkach wartość ze znakiem jest traktowana jako jej odpowiednik bez znaku, co oznacza, że liczba ujemna staje się dużą liczbą dodatnią. Rozważ następującą „naprawę" opisanego wcześniej przepełnienia stosu:

```
bool CheckLogin(char* username, int len, char* password)

{

char _username[32] = “”;

if (len < 32)

strncpy(_username, username, len);

...
```

Tutaj funkcja przyjmuje zarówno nazwę użytkownika podaną przez użytkownika, jak i liczbę całkowitą ze znakiem wskazującą jej długość. Programista tworzy bufor o stałym rozmiarze na stosie i sprawdza, czy długość jest mniejsza niż rozmiar bufora. Jeśli tak, programista wykonuje zliczoną kopię bufora, mającą na celu zapewnienie, że bufor nie zostanie przepełniony. Jeśli parametr len jest liczbą dodatnią, ten kod zachowuje się zgodnie z przeznaczeniem. Jeśli jednak atakujący może spowodować przekazanie do funkcji wartości ujemnej, test ochronny programisty zostaje obalony. Porównanie z 32 nadal się udaje, ponieważ kompilator traktuje obie liczby jako liczby całkowite ze znakiem. W związku z tym ujemna długość jest przekazywana do funkcji strncpy jako jej parametr count. Ponieważ strncpy przyjmuje jako ten parametr liczbę całkowitą bez znaku, kompilator niejawnie rzutuje wartość len na ten typ, więc wartość ujemna jest traktowana jako duża liczba dodatnia. Jeśli ciąg nazwy użytkownika dostarczony przez użytkownika jest dłuższy niż 32 bajty, bufor jest przepełniany, tak jak w przypadku standardowego przepełnienia stosu. Ten rodzaj ataku jest zwykle możliwy tylko wtedy, gdy atakujący może bezpośrednio kontrolować parametr długości. Na przykład, być może jest obliczany przez JavaScript po stronie klienta i przesyłany z żądaniem wraz z ciągiem znaków, do którego się odnosi. Jeśli jednak zmienna całkowita jest wystarczająco mała (na przykład krótka), a program oblicza długość po stronie serwera, osoba atakująca może również wprowadzić wartość ujemną za pomocą przepełnienia liczby całkowitej, przesyłając zbyt długi ciąg do aplikacja.

Wykrywanie luk w zabezpieczeniach liczb całkowitych

Oczywiście głównymi lokalizacjami, w których należy szukać luk w zabezpieczeniach dotyczących liczb całkowitych, są wszelkie przypadki, w których wartość całkowita jest przesyłana od klienta do serwera. Takie zachowanie zwykle pojawia się na dwa różne sposoby:

* Aplikacja może przekazywać wartości całkowite w normalny sposób jako parametry w ciągu zapytania, plikach cookie lub treści wiadomości. Liczby te są zwykle przedstawiane w postaci dziesiętnej przy użyciu standardowych znaków ASCII. Najbardziej prawdopodobnymi celami testowania są pola, które wydają się reprezentować długość przesyłanego łańcucha.

* Aplikacja może przekazywać wartości całkowite osadzone w większym blobie danych binarnych. Te dane mogą pochodzić z komponentu po stronie klienta, takiego jak formant ActiveX, lub mogły zostać przesłane przez klienta w ukrytym polu formularza lub pliku cookie. Liczby całkowite związane z długością mogą być trudniejsze do zidentyfikowania w tym kontekście. Zazwyczaj są one reprezentowane w postaci szesnastkowej i często bezpośrednio poprzedzają ciąg lub bufor, do którego się odnoszą. Należy pamiętać, że dane binarne mogą być kodowane za pomocą Base64 lub podobnych schematów do transmisji przez HTTP.

KROKI HACKOWANIA

1. Po zidentyfikowaniu celów do testów należy wysłać odpowiednie ładunki zaprojektowane tak, aby uruchamiały wszelkie luki w zabezpieczeniach. Dla każdego docelowego elementu danych wyślij po

kolei serię różnych wartości reprezentujących przypadki graniczne dla wersji ze znakiem i bez znaku o różnych rozmiarach liczb całkowitych. Na przykład:

* 0x7f i 0x80 (127 i 128)

* 0xff i 0x100 (255 i 256)

* 0x7fff i 0x8000 (32767 i 32768)

* 0xffff i 0x10000 (65535 i 65536)

* 0x7fffffff i 0x80000000 (2147483647 i 2147483648)

* 0xffffffff i 0x0 (4294967295 i 0)

2. Gdy modyfikowane dane są reprezentowane w postaci szesnastkowej, należy wysłać wersje little-endian oraz big-endian każdego przypadku testowego — na przykład ff7f oraz 7fff. Jeśli liczby szesnastkowe są przesyłane w postaci ASCII, należy użyć takiej samej wielkości liter, jakiej sama aplikacja używa dla znaków alfabetu, aby upewnić się, że są one poprawnie dekodowane.

3. Należy monitorować reakcje aplikacji na nietypowe zdarzenia w taki sam sposób, jak opisano w przypadku luk w zabezpieczeniach związanych z przepełnieniem bufora.

Luki w zabezpieczeniach ciągów formatu

Luki w zabezpieczeniach ciągów formatujących pojawiają się, gdy kontrolowane przez użytkownika dane wejściowe są przekazywane jako parametr ciągu formatującego do funkcji, która przyjmuje specyfikatory formatu, które mogą być niewłaściwie używane, jak w rodzinie funkcji printf w C. Funkcje te przyjmują zmienną liczbę parametrów, które mogą składają się z różnych typów danych, takich jak liczby i łańcuchy. Ciąg formatu przekazany do funkcji zawiera specyfikatory, które mówią jej, jakie dane są zawarte w parametrach zmiennych iw jakim formacie powinny być renderowane. Na przykład poniższy kod wyświetla komunikat zawierający wartość zmiennej count w postaci ułamka dziesiętnego:

printf(“The value of count is %d”, count.);

Najbardziej niebezpiecznym specyfikatorem formatu jest %n. Nie powoduje to drukowania żadnych danych. Powoduje raczej, że liczba bajtów danych wyjściowych do tej pory jest zapisywana pod adresem wskaźnika przekazanego jako powiązany parametr zmiennej. Na przykład:

int count = 43;

int written = 0;

printf(“The value of count is %d%n.\n”, count, &written.);

printf(“%d bytes were printed.\n”, written);

wyprowadza następujące informacje:

The value of count is 43.

24 bytes were printed.

Jeśli ciąg formatu zawiera więcej specyfikatorów niż liczba przekazanych parametrów zmiennych, funkcja nie ma możliwości wykrycia tego, więc po prostu kontynuuje przetwarzanie parametrów ze stosu wywołań. Jeśli atakujący kontroluje całość lub część ciągu formatującego przekazanego do funkcji printfstyle, zwykle może to wykorzystać do nadpisania krytycznych części pamięci procesu i ostatecznie

spowodować wykonanie dowolnego kodu. Ponieważ atakujący kontroluje ciąg formatu, może kontrolować zarówno liczbę bajtów wyprowadzanych przez funkcję, jak i wskaźnik na stosie, który jest zastępowany liczbą bajtów danych wyjściowych. Umożliwia mu to nadpisanie zapisanego adresu zwrotnego lub wskaźnika do procedury obsługi wyjątków i przejęcie kontroli nad wykonaniem w taki sam sposób, jak w przypadku przepełnienia stosu.

Wykrywanie luk w zabezpieczeniach ciągów formatujących

Najbardziej niezawodnym sposobem wykrywania błędów ciągów formatujących w aplikacji zdalnej jest przesyłanie danych zawierających różne specyfikatory formatu i monitorowanie wszelkich anomalii w zachowaniu aplikacji. Podobnie jak w przypadku niekontrolowanego wyzwalania luk w zabezpieczeniach związanych z przepełnieniem bufora, prawdopodobne jest, że sondowanie w poszukiwaniu błędów ciągu formatującego spowoduje awarię podatnej aplikacji.

KROKI HACKOWANIA

1. Kierując po kolei każdy parametr, prześlij ciągi zawierające duże liczby specyfikatorów formatu %n i %s:

%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n

%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s

Należy zauważyć, że niektóre operacje na łańcuchu formatu mogą ignorować specyfikator %n ze względów bezpieczeństwa. Podanie zamiast tego specyfikatora %s powoduje, że funkcja usuwa odwołania do każdego parametru na stosie, co prawdopodobnie skutkuje naruszeniem zasad dostępu, jeśli aplikacja jest podatna na ataki.

2. Funkcja Windows FormatMessage używa specyfikatorów w inny sposób niż rodzina printf. Aby przetestować podatne na ataki wywołania tej funkcji, należy użyć następujących ciągów znaków:

%1!n!%2!n!%3!n!%4!n!%5!n!%6!n!%7!n!%8!n!%9!n!%10!n! itp...

%1!s!%2!s!%3!s!%4!s!%5!s!%6!s!%7!s!%8!s!%9!s!%10!s! itp...

3. Pamiętaj, aby w adresie URL zakodować znak % jako %25.

4. Należy monitorować reakcje aplikacji na nietypowe zdarzenia w taki sam sposób, jak w przypadku luk w zabezpieczeniach związanych z przepełnieniem bufora.

Streszczenie

Luki w oprogramowaniu w natywnym kodzie stanowią stosunkowo niszowy obszar w odniesieniu do ataków na aplikacje internetowe. Większość aplikacji działa w zarządzanym środowisku wykonawczym, w którym nie występują klasyczne wady oprogramowania opisane w tej Części. Jednak czasami tego rodzaju luki są bardzo istotne i stwierdzono, że wpływają na wiele aplikacji internetowych działających na urządzeniach sprzętowych i innych niezarządzanych środowiskach. Dużą część takich luk można wykryć, przesyłając określony zestaw przypadków testowych na serwer i monitorując jego zachowanie. Niektóre luki w aplikacjach natywnych są stosunkowo łatwe do wykorzystania, na przykład opisana w tym rozdziale luka „off-by-one”. Jednak w większości przypadków trudno je wykorzystać, mając jedynie zdalny dostęp do podatnej aplikacji. W przeciwieństwie do większości innych rodzajów luk w zabezpieczeniach aplikacji internetowych, nawet samo szukanie klasycznych wad oprogramowania może spowodować stan odmowy usługi, jeśli aplikacja jest podatna na ataki. Przed wykonaniem takich testów należy upewnić się, że aplikacja akceptuje związane z tym ryzyko.

Pytania

1. Jeśli nie ma żadnych specjalnych zabezpieczeń, dlaczego przepełnienia bufora oparte na stosie są generalnie łatwiejsze do wykorzystania niż przepełnienia sterty?

2. W jaki sposób w językach C i C++ określa się długość łańcucha?

3. Dlaczego luka w zabezpieczeniach zwykłego urządzenia sieciowego związana z przepełnieniem bufora zwykle wiąże się ze znacznie większym prawdopodobieństwem wykorzystania niż przepełnienie zastrzeżonej aplikacji internetowej działającej w Internecie?

4. Dlaczego poniższy ciąg rozmyty miałby nie identyfikować wielu przypadków luk w zabezpieczeniach ciągów formatujących?

%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n %n%n%n%n%n...

5. Poszukujesz luk w zabezpieczeniach związanych z przepełnieniem bufora w aplikacji internetowej, która intensywnie wykorzystuje natywne komponenty kodu. Znajdujesz żądanie, które może zawierać lukę w jednym z parametrów; jednak anomalne zachowanie, które zaobserwowałeś, jest trudne do wiarygodnego odtworzenia. Czasami przesłanie długiej wartości powoduje natychmiastową awarię. Czasami trzeba przesłać go kilka razy z rzędu, aby spowodować awarię. Czasami po dużej liczbie łagodnych żądań dochodzi do awarii. Jaka jest najbardziej prawdopodobna przyczyna zachowania aplikacji?