Audyt binarny: hakowanie oprogramowania o zamkniętym kodzie źródłowym

Wiele krytycznych dla bezpieczeństwa i szeroko wdrażanych baz kodu to źródła zamknięte, w tym niektóre z dominujących rodzin systemów operacyjnych zarówno dla serwerów, jak i komputerów stacjonarnych. Aby ocenić bezpieczeństwo oprogramowania o zamkniętym kodzie źródłowym, wykraczające poza możliwości testowania rozmytego, niezbędny jest audyt binarny. Ogólnie rzecz biorąc, audyt binarny jest uważany za trudniejszy niż audyt za pomocą kodu źródłowego. Chociaż może to wydawać się złą wiadomością dla początkujących, można to również uznać za korzyść. Obecnie znacznie mniej osób przeprowadza audyt plików binarnych, a mniej oczu ułatwia pracę. Wiele klas błędów, które praktycznie wymarły w oprogramowaniu open source, nadal pozostaje w komercyjnych bazach kodu zamkniętego. Audytowanie plików binarnych jest nadal nauką niedoskonałą, a wiele rzeczy, które można dość łatwo zweryfikować podczas audytu kodu źródłowego, jest odwrotnie dość trudnych do ustalenia podczas badania pliku binarnego. Dzięki praktyce i przy pomocy kilku przydatnych narzędzi, większość frustracji związanych z audytem binarnym może zostać usunięta. Wielu badaczy bezpieczeństwa nie wychodzi poza ograniczenia testowania rozmytego podczas audytu oprogramowania komercyjnego. Chociaż testy fuzz dowiodły, że mogą ujawnić błędy w oprogramowaniu, tak naprawdę nie można w rozsądnym czasie przenieść wszystkich możliwych wzorców wejściowych do żadnego dużego oprogramowania. Audyt binarny może zaoferować pełniejszy obraz wewnętrznego działania aplikacji i luk w zabezpieczeniach, które może zawierać.

Audyt binarny a audyt kodu źródłowego: oczywiste różnice

Audyt binarny można porównać do audytu kodu źródłowego, ponieważ szukasz tych samych klas błędów i wad w oprogramowaniu; zmienił się jednak sposób ich poszukiwania. Jeśli jesteś już zaznajomiony z audytem kodu źródłowego, prawdopodobnie nie będziesz musiał zbytnio zmieniać swojego procesu myślowego. Jednak twoja metodologia ulegnie zmianie. Przede wszystkim będziesz potrzebować doskonałego zrozumienia języka asemblera związanego z platformą, na której twój plik binarny będzie działał. Jeśli nie masz jasności co do ważnych instrukcji, prawdopodobnie błędnie zinterpretujesz większość przeczytanego kodu i skończysz zdezorientowany i sfrustrowany. Jeśli nie jesteś w stanie przeczytać i zrozumieć dezasemblacji, dokładna nauka odpowiedniego języka asemblera jest dobrym miejscem do rozpoczęcia. Niektóre pliki binarne, zwłaszcza pliki binarne kodu p, takie jak klasy Java lub aplikacje Visual Basic, mogą być w pełni dekompilowane do czegoś, co bardzo przypomina ich oryginalny kod źródłowy. Jednak większości plików binarnych nie można wiarygodnie zdekompilować za pomocą dzisiejszych narzędzi. Ten rozdział skupia się na kontroli plików binarnych Intel x86, zwłaszcza tych skompilowanych za pomocą kompilatora Microsoft Visual C++. Podczas audytu pliku binarnego, tak jak podczas audytu kodu źródłowego, najważniejsze jest zrozumienie kodu, który czytasz. Jednak to, co może być bardzo oczywistym sprawdzeniem bezpieczeństwa w źródle, może często przekładać się na jedną lub dwie instrukcje. Dlatego zdecydowanie konieczne jest bycie świadomym wykonywania programu w dowolnym momencie funkcji. Na przykład często trzeba wiedzieć, jakie wartości są przechowywane w jakich rejestrach w określonym momencie wykonywania, a wiele wartości może być wymienianych do i z określonego rejestru w dowolnym bloku kodu. Niektóre luki są równie łatwe lub łatwiejsze do wykrycia w pliku binarnym niż w kodzie źródłowym; jednak większość błędów będzie bardziej subtelna i trudniejsza do wykrycia dla kogoś, kto próbuje przeprowadzić swój pierwszy audyt binarny. Gdy zaznajomisz się z kodem tworzonym przez niektóre kompilatory, audyt plików binarnych stanie się prawie tak prosty, jak audyt kodu źródłowego.

IDA Pro - narzędzie handlu

Interactive Disassembler Pro, znany powszechnie jako IDA Pro, jest powszechnie uznawany za najlepsze narzędzie do analizy lub audytu plików binarnych. Jest rozwijany i sprzedawany przez

belgijską firmę Datarescue (www.datarescue.com) i jest dostępny za rozsądną cenę. Jeśli będziesz wykonywać dużo audytów binarnych, powinieneś poważnie rozważyć zakup licencji. Chociaż IDA Pro ma swoje braki, nadal jest bardzo dobrym deasemblerem i daleko wyprzedza konkurencję. IDA Pro obsługuje wiele różnych formatów binarnych na wielu platformach i najprawdopodobniej będzie obsługiwać nawet najbardziej niejasne formaty, które chcesz rozmontować. Przechowuje dane wyjściowe zdeasemblowanego programu w formacie bazy danych i pozwala na nazywanie i zmianę nazwy praktycznie każdego aspektu analizowanego programu. Komentarze linijka po linijce to często spotykana funkcja pomocna, gdy próbujesz analizować złożone konstrukcje kodu. Podobnie jak wiele deasemblerów, IDA Pro może wyświetlać ciągi i odsyłacze do większości fragmentów kodu lub danych.

Funkcje : Szybki kurs awaryjny

Podstawowe zrozumienie funkcji IDA Pro ogromnie pomoże w każdej analizie binarnej. Oczywiście nie jest konieczne zrozumienie wszystkich zaawansowanych funkcji, aby rozpocząć audyt plików binarnych. W głównym widoku IDA Pro (Widok-A) można znaleźć większość potrzebnych informacji. To jest widok demontażu i zawiera zdeasemblowaną reprezentację analizowanego kodu. Wyświetlacz jest oznaczony kolorami, aby ułatwić przeglądanie. Wartości stałe są zielone, nazwane wartości są niebieskie, importowane funkcje są różowe, a większość kodu jest ciemnoniebieska. Możesz także podświetlić konkretny ciąg na żółto, umieszczając nad nim kursor (jest to bardzo przydatne, gdy próbujesz zlokalizować odniesienia do konkretnego adresu lub zarejestrować się w dużym bloku kodu). Główny widok pokaże kod na zasadzie funkcja po funkcji. Regiony kodu, które należą do prawidłowych funkcji, mają adresy oznaczone kolorem czarnym, a regiony kodu, które nie należą do żadnej funkcji, są brązowe. Importowane adresy danych (IAT lub idata) są różowe, dane tylko do odczytu są szare, a dane do zapisu są żółte. IDA Pro ma widok szesnastkowy, w którym można wyświetlić szesnastkową i ciągową reprezentację kodu. Okno nazw wyświetla wszystkie nazwane lokalizacje w aplikacji, okno funkcji wyświetla wszystkie znalezione funkcje, a okno ciągów wyświetla wszystkie znane ciągi w programie. Istnieją inne okna, takie jak te służące do tworzenia list struktur i wyliczeń. W tych oknach można znaleźć większość potrzebnych informacji. IDA Pro będzie przechowywać odwołania krzyżowe do kodu, na który wskazują wszelkie skoki, wywołania lub odwołania do danych. Jest to przydatne podczas śledzenia wstecz przepływu wykonywania z dowolnej lokalizacji. Będzie również próbował zinterpretować układ lokalnego stosu dla dowolnej funkcji. IDA Pro zrobi to poprawnie dla funkcji ze standardową ramką stosu, ale czasami ma problemy z funkcjami, które zoptymalizowały wskaźnik ramki. IDA Pro ma możliwość nazwania dowolnej lokalizacji w programie i wprowadzania komentarzy w dowolnej lokalizacji. To sprawia, że analiza kodu jest znacznie prostsza i może znacznie ułatwić powrót do fragmentu kodu następnego dnia i wciąż pamiętanie, co się dzieje. IDA Pro ma również wbudowany kod od wersji 4.2, który może przedstawiać kod graficznie. W wielu przypadkach okazało się to bardzo przydatne. Istnieje kilka wtyczek innych firm do IDA Pro, które również mogą być przydatne, ale większość z nich nie jest specjalnie zaprojektowana do audytu plików binarnych. Możliwe jest określenie typu danych znajdujących się w dowolnej lokalizacji w pamięci. Chociaż IDA Pro spróbuje odgadnąć najlepiej, jak potrafi, czy określony adres zawiera kod, dane binarne, dane ciągów lub inne formaty, nie zawsze może to zrobić poprawnie. Użytkownik ma możliwość zmiany wszystkiego, co może nie wyglądać dobrze.

Symbole debugowania

Firma Microsoft oferuje informacje o symbolach do pobrania dla każdej większej wersji swoich systemów operacyjnych. Pakiety symboli systemu Windows można pobrać ze strony Windows Hardware and Driver Central w witrynie Microsoft.com (www.microsoft.com/whdc/hwdev/) i są niezwykle przydatne podczas analizowania plików binarnych. Symbole są zazwyczaj dystrybuowane w postaci pliku PDB, który jest formatem bazy danych programu generowanym przez MSVC++. Pliki te

zawierają co najmniej nazwy funkcji dla prawie każdej funkcji i statycznej lokalizacji danych w pliku binarnym. W przypadku niektórych plików binarnych pliki PDB będą zawierać nieudokumentowane struktury wewnętrzne i nazwy zmiennych lokalnych. Binarny jest zaskakująco łatwiejszy do zrozumienia, gdy wszystko ma nazwy. Pakiety symboli są dystrybuowane przez firmę Microsoft jako dodatki Service Pack i nie są ogólnie dostępne dla poprawek hot fix. Prawie każda aplikacja, biblioteka i sterownik w podstawowym systemie operacyjnym będą miały publicznie dostępne symbole. IDA Pro może importować pliki PDB i zmieniać nazwy wszystkich funkcji w formacie binarnym. Ponadto narzędzia innych firm, takie jak PdbDump, mogą interpretować pliki PDB i wydobywać przydatne informacje.

Wprowadzenie do audytu binarnego

Aby pomyślnie przeprowadzić audyt plików binarnych, musisz poprawnie i dokładnie zrozumieć kod wygenerowany przez kompilator. Istnieje wiele konstrukcji kodu kompilatora, których cele nie są intuicyjne ani od razu oczywiste. W tym rozdziale próbujemy wprowadzić audytorów binarnych do większości standardowych konstrukcji kodu, a także niestandardowych konstrukcji kodu, które często występują w kodzie, z nadzieją, że skompilowany kod będzie prawie tak łatwy do zrozumienia jak kod źródłowy.

Ramki stosu

Zrozumienie układu ramki stosu dowolnej funkcji znacznie ułatwi zrozumienie kodu, a w niektórych przypadkach znacznie ułatwi również określenie, czy istnieje przepełnienie stosu. Chociaż istnieje kilka typowych układów ramek stosu na x86, nic nie jest ustandaryzowane, a układ jest określany przez kompilator. Bardziej typowe przykłady są omówione tutaj.

Tradycyjne ramki stosu oparte na BP

Najpopularniejszym układem ramki stosu dla funkcji jest tradycyjna ramka oparta na BP, w której rejestr wskaźnika ramki, EBP, jest stałym wskaźnikiem do poprzedniej ramki stosu. Wskaźnik ramki jest również stałą lokalizacją, względem której dostępne są argumenty funkcji i lokalne zmienne stosu. Wstęp do funkcji korzystającej z tej tradycyjnej ramki stosu wygląda następująco w notacji Intela:

push ebp // zapisz stary wskaźnik ramki na stosie

mov ebp, esp // ustaw wskaźnik nowej ramki na esp

sub esp, 5ch // rezerwuj miejsce na zmienne lokalne

W tym momencie lokalne zmienne stosu znajdują się w ujemnym przesunięciu względem EBP, a argumenty funkcji znajdują się w dodatnim przesunięciu. Pierwszy argument funkcji znajduje się w EBP + 8. IDA Pro zmieni nazwę lokalizacji EBP + 8 na EBP + arg_0. Prawie wszystkie odwołania do argumentów i lokalnych zmiennych stosu będą tworzone względem wskaźnika ramki w funkcjach z tym typem ramki. Ten układ stosu został bardzo dobrze udokumentowany i jest najłatwiejszy do naśladowania podczas audytu. Większość kodu generowanego przez MSVC++ i przez gcc użyje tej ramki stosu.

Funkcje bez wskaźnika ramki

Ze względu na optymalizację wiele kompilatorów wygeneruje kod, który optymalizuje użycie wskaźnika ramki. Niektóre kompilatory mogą nawet w niektórych przypadkach używać rejestru wskaźnika ramki jako rejestru ogólnego przeznaczenia. W takim przypadku funkcja uzyska dostęp do swoich argumentów i zmiennych lokalnych odnoszących się do wskaźnika ESP stosu zamiast wskaźnika ramki.

Chociaż wskaźnik ramki w tradycyjnej ramce stosu jest stały, wskaźnik stosu przesuwa się po całej lokalizacji funkcji, zmieniająca się za każdym razem, gdy operacja wypchnie lub zdejmie coś ze stosu. Poniższy przykład próbuje to zilustrować :

this_function:

push esi

push edi

push ebx

push dword ptr [esp+10h] // first argument to this_function

push dword ptr [esp+18h] // second argument to this_function

call some_function

Gdy funkcja jest wprowadzana po raz pierwszy, pierwszym argumentem jest ESP+4. Po zapisaniu trzech rejestrów pierwszy argument ma teraz ESP+10h. Po przesunięciu pierwszego argumentu funkcji jako parametru do some_function drugi argument funkcji znajduje się teraz pod adresem ESP+18h. IDA Pro podejmuje próbę określenia położenia wskaźnika stosu w dowolnym miejscu w funkcji. W ten sposób próbuje zidentyfikować, do czego naprawdę odnoszą się dostępy do danych względnych wskaźnika stosu. Jednak gdy nie zna konwencji wywoływania używanych przez funkcje zewnętrzne, IDA Pro może się pomylić i stworzyć bardzo mylący dezasembler. Czasami może być konieczne ręczne obliczenie położenia wskaźnika stosu w określonym punkcie funkcji w celu określenia rozmiaru buforów stosu. Na szczęście to zamieszanie nie zdarza się zbyt często.

Nietradycyjne ramki stosu oparte na BP

Microsoft Visual Studio .NET 2003 od czasu do czasu tworzy kod z ramką stosu, która korzysta ze stałego wskaźnika ramki, choć nie w tradycyjnym sensie. Gdy wskaźnik ramki jest stały i cały dostęp do argumentów i lokalnych zmiennych stosu jest do niego zależny, nie wskazuje on wskaźnika ramki wywołującej funkcji, ale raczej lokalizację w ujemnym przesunięciu od miejsca, w którym byłby tradycyjny wskaźnik ramki. Przykładowy prolog funkcji może wyglądać następująco:

push ebp

lea ebp, [esp-5ch]

sub esp, 98h

Pierwszy argument funkcji byłby umieszczony w EBP+64h, zamiast tradycyjnej lokalizacji EBP+8. Zakres pamięci od EBP-3ch do EBP+5ch byłby zajęty przez lokalne zmienne stosu. System operacyjny Windows Server 2003 został skompilowany z kodem zawierającym tę nietradycyjną ramkę opartą na BP i można go znaleźć w bibliotekach i usługach systemowych. W chwili pisania tego tekstu IDA Pro nie rozpoznaje tej konstrukcji kodu i całkowicie błędnie zinterpretuje lokalną ramkę stosu dla funkcji tego typu. Mam nadzieję, że wsparcie dla tego dziwactwa kompilatora zostanie dodane w najbliższej przyszłości.

Wywoływanie konwencji

Różne funkcje w aplikacji mogą używać różnych konwencji wywoływania, zwłaszcza jeśli części aplikacji zostały napisane w różnych językach. Przydatne jest zrozumienie różnych konwencji wywoływania występujących w językach opartych na C. Ogólnie rzecz biorąc, tylko dwie konwencje wywoływania będą powszechnie widoczne w kodzie C lub C++ generowanym przez MSVC++ lub gcc.

Konwencja C Calling

Konwencja wywoływania C nie tylko odnosi się do kodu C, ale jest sposobem przekazywania argumentów i przywracania stosu programu. Przy tej konwencji wywoływania argumenty funkcji są odkładane na stos od prawej do lewej, gdy pojawiają się w kodzie źródłowym. Innymi słowy, ostatni argument jest odkładany jako pierwszy, a pierwszy argument jest ostatnim odkładanym przed wywołaniem funkcji. Od funkcji wywołującej zależy przywrócenie wskaźnika stosu po powrocie wywołania. Przykładem konwencji wywoływania języka C jest:

some_function(some_pointer,some_integer);

This function call would look something like the following when using the C calling convention:

push some_integer

push some_pointer

call some_function

add esp, 8

Zauważ, że drugi argument funkcji jest wstawiany przed pierwszym i że wskaźnik stosu jest przywracany przez funkcję wywołującą. Ponieważ ta funkcja miała dwa argumenty, wskaźnik stosu musiał zostać zwiększony o 8 bajtów. Często zdarza się również, że stos jest przywracany przy użyciu instrukcji POP x86 z miejscem docelowym rejestru zdrapek. W tym przykładzie byłoby możliwe przywrócić stos, wykonując dwukrotnie POP ECX, za każdym razem przywracając 4 bajty.

Konwencja połączeń Stdcall

Inną konwencją wywoływania często spotykaną w kodzie C i C++ jest Stdcall. Argumenty są przekazywane w tej samej kolejności, co w konwencji wywoływania C, przy czym pierwszy argument funkcji jest odkładany na stos jako ostatni przed wywołaniem funkcji. Jednak generalnie od wywoływanej funkcji zależy odtworzenie stosu. Zwykle odbywa się to na x86 za pomocą instrukcji return, która zwalnia miejsce na stosie. Na przykład funkcja, która ma trzy argumenty i używa konwencji wywołania Stdcall, zwróciłaby się z RET 0Ch, zwalniając po powrocie 12 bajtów ze stosu. Stdcall jest ogólnie bardziej wydajny, ponieważ funkcja wywołująca nie musi zwalniać miejsca na stosie. Funkcje, które akceptują zmienną liczbę argumentów, takie jak funkcje typu printf, nie mogą zwolnić miejsca na stosie zajmowanego przez ich argumenty. Musi to zrobić funkcja wywołująca, która ma wiedzę o tym, ile istnieje argumentów.

Kod generowany przez kompilator

Kompilatory mogą generować dużo kodu, który na pierwszy rzut oka może być mylący. Przyjrzyjmy się niektórym typowym obszarom, w których kompilator będzie dodawał instrukcje, oraz sposobom rozpoznawania struktur generowanych przez kompilator.

Układy funkcji

Układ kodu generowanego przez kompilator w funkcji jest nieco zmienny. Funkcja zazwyczaj zaczyna się prologiem funkcji, a kończy epilogiem funkcji i zwrotem. Jednak funkcja niekoniecznie musi kończyć się zwrotem i dość często można zobaczyć funkcję z kodem po instrukcji powrotu. Ten kod w końcu przeskoczy z powrotem do instrukcji return. Chociaż funkcja może zwracać się w wielu miejscach, kompilator zoptymalizuje zdolność funkcji do przeskakiwania do jednego wspólnego miejsca zwrotu. Od wersji Visual Studio 6 kompilator MSVC++ generował kod z bardzo niekonwencjonalnymi układami

funkcji. Kompilator używa pewnej logiki, aby określić, które gałęzie prawdopodobnie zostaną pobrane, a które są mniej prawdopodobne. Te uznane za mniej prawdopodobne są usuwane z linii funkcji głównej i umieszczane jako fragmenty kodu w odległych lokalizacjach pamięci. Te fragmenty kodu to często kod, który radzi sobie z nietypowymi błędami lub mało prawdopodobnymi scenariuszami. Jednak w tych fragmentach kodu często występują luki w zabezpieczeniach i należy je przeglądać podczas audytu plików binarnych. Te fragmenty kodu są często wskazywane czerwonymi strzałkami skoku w IDA Pro i przez wiele lat były powszechną częścią skompilowanego kodu MSVC++. IDA Pro może nie radzić sobie poprawnie z tymi fragmentami kodu i może nie odnotowywać w nich dostępu do zmiennych stosu lokalnego lub nie wyświetlać ich poprawnie. W wysoce zoptymalizowanym kodzie kilka funkcji może współdzielić fragmenty kodu. Na przykład, jeśli kilka funkcji powraca w ten sam sposób i przywraca te same rejestry i miejsce na stosie, technicznie możliwe jest, aby współdzieliły ten sam epilog funkcji i kod powrotu. Jest to jednak dość rzadkie i tak naprawdę było widoczne tylko w NTDLL w systemach operacyjnych Windows NT.

Instrukcja If

Instrukcje if są jedną z najczęstszych konstrukcji kodu C i czasami bardzo łatwo je zobaczyć i zinterpretować w skompilowanym kodzie. Najczęściej są one reprezentowane przez instrukcje CMP lub TEST, po których następuje skok warunkowy. Poniższy przykład przedstawia prostą instrukcję C if i odpowiadającą jej reprezentację zestawu. Kod C:

```
int some_int;

if(some_int != 32)

some_int = 32;
```

Compiled Representation (ebp-4 = some_int):

```
mov eax, [ebp-4]

cmp eax, 32

jnz past_next_instruction

mov eax, 32
```

Jeśli stwierdzenia generalnie charakteryzują się skokami do przodu lub rozgałęzieniami; jednak niekoniecznie jest to prawdą, a reorganizacja kodu przez kompilator może spowodować spustoszenie w tym problemie. W niektórych kontekstach będzie bardzo oczywiste, że gałąź warunkowa była instrukcją if, ale w innych kontekstach instrukcje if są trudne do odróżnienia od innych konstrukcji kodu, takich jak pętle. Lepsze zrozumienie ogólnej struktury funkcji powinno wyjaśnić, gdzie znajdują się stwierdzenia.

Pętle For i While

Konstrukcje pętli w aplikacji są bardzo powszechnym miejscem znajdowania luk w zabezpieczeniach. Rozpoznawanie ich w plikach binarnych jest często kluczową częścią audytu. Chociaż w skompilowanym kodzie nie da się całkowicie odróżnić od siebie różnych typów pętli, ich funkcjonalne rozpoznanie w plikach binarnych jest zwykle dość proste. Zazwyczaj charakteryzują się odgałęzieniem wstecz lub skokiem, który prowadzi do powtarzającej się sekcji kodu. Poniższy przykład ilustruje prostą pętlę while i jej skompilowaną reprezentację. Kod C:

```
char *ptr,*output,*outputend;
```

```
while(*ptr) {

*output++ = *ptr++;

if(output >= outputend)

break;
```

Compiled Representation (ecx = ptr, edx = output, ebp+8 = outputend):

```
mov al, [ecx]

test al, al

jz loop_end

mov [edx], al

inc ecx

inc edx

cmp edx, [ebp+8]

jae loop_end

jmp loop_begin
```

Funkcjonalnie kod mógł być taki sam, jak prosta pętla for, co utrudnia określenie, jaki rodzaj instrukcji znajdował się w oryginalnym kodzie źródłowym. Jednak funkcjonalność kodu jest ważniejsza niż jego pierwotny stan jako kod źródłowy, a pętle, takie jak ta pokazana tutaj, są źródłem wielu błędów w aplikacjach z zamkniętym kodem źródłowym.

Instrukcja Switch

Instrukcje switch są ogólnie dość złożonymi konstrukcjami w kodzie asemblera i czasami mogą prowadzić do skompilowanego kodu, który wygląda trochę dziwnie. W zależności od kompilatora i rzeczywistej instrukcji switch, skonstruowany kod może znacznie różnić się strukturą. Instrukcja switch może być nieefektywnie podzielona na kilka instrukcji if, a niektóre kompilatory zrobią to w pewnych sytuacjach. Same stwierdzenia mogą być prostsze do zrozumienia, a audytor czytający kod może nigdy nie podejrzewać, że dany kod kiedykolwiek był czymś innym niż grupą następujących po sobie stwierdzeń. Jeśli przypadki przełączników są sekwencyjne, kompilator często generuje tabelę skoków i indeksuje ją z przypadkiem przełącznika. Jest to bardzo skuteczny sposób radzenia sobie z przełącznikami z przypadkami sekwencyjnymi, ale nie zawsze jest to możliwe. Przykład może wyglądać następująco. Kod C:

```
int some_int,other_int;

switch(some_int) {

case 0:

other_int = 0;

break;

case 1:
```

```
other_int =10;
break;
case 2:
other_int = 30;
break;
default:
other_int = 50;
break;
}
```

Compiled Representation (some_int = eax, other_int = ebx):

```
cmp eax, 2
ja default_case
jmp switch_jmp_table[eax*4];
case_0:
xor ebx, ebx
jmp end_switch
case_1:
mov ebx, 10
jmp end_switch
case 2:
mov ebx, 30
jmp end_switch
default_case:
mov ebx, 50
end_switch:
```

W miejscu pamięci tylko do odczytu zostanie znaleziona tabela danych switch_jmp_table zawierająca kolejno przesunięcia case_0, case_1 i case_2. IDA Pro wykonuje bardzo dobrą robotę w wykrywaniu instrukcji przełącznika skonstruowanych tak, jak pokazano, i bardzo dokładnie wskazuje użytkownikowi, które przypadki zostałyby wywołane przez które wartości. W przypadku, gdy wartości przypadków przełączników nie są uporządkowane sekwencyjnie, nie można ich łatwo i efektywnie wykorzystać jako indeksu w tabeli skoków. W tym momencie kompilatory często używają konstrukcji, w której wartość przełącznika jest zmniejszana lub odejmowana, aż do osiągnięcia wartości zero odpowiadającej wartości wielkości przełącznika. Dzięki temu instrukcja switch może wydajnie radzić

sobie z wartościami wielkości liter, które są odległe liczbowo. Na przykład, jeśli instrukcja switch miała zajmować się wartościami case 3, 4, 7 i 24, może to zrobić w następujący sposób (EAX = wartość case):

```
sub eax, 3

jz case_three

dec eax

jz case_four

sub eax, 3

jz case_seven

sub eax, 17

jz case_twenty_four

jmp default
```

Ten kod poradziłby sobie poprawnie ze wszystkimi możliwymi przypadkami przełączania, a także z wartościami domyślnymi i jest często spotykany w kodzie generowanym przez nowoczesne kompilatory MSVC++.

Konstrukcje kodu podobne do memcpy

Wiele kompilatorów zoptymalizuje funkcję biblioteki memcpy do prostych instrukcji asemblera, które są znacznie wydajniejsze niż wywołanie funkcji. Ten rodzaj operacji kopiowania pamięci może potencjalnie być źródłem luk w zabezpieczeniach związanych z przepełnieniem bufora i może być łatwo rozpoznany podczas demontażu. Zastosowany zestaw instrukcji jest następujący:

```
mov esi, source_address

mov ebx, ecx

shr ecx, 2 // length divided by four

mov edi, eax // destination address

repe movsd // copy four byte blocks

mov ecx, ebx

and ecx, 3 // remainder size

repe movsb // copy it
```

W takim przypadku dane są kopiowane z rejestru źródłowego ESI do rejestru docelowego EDI. Dane są początkowo kopiowane w 4-bajtowych blokach ze względu na szybkość przez instrukcję repe movsd. To kopiuje liczbę ECX 4-bajtowych bloków z ESI do EDI, dlatego długość w ECX jest dzielona przez 4. Instrukcja repe movsb kopiuje resztę danych. memset jest często optymalizowany dokładnie w ten sam sposób przy użyciu instrukcji repestosd z rejestrem AL przechowującym znak do memset. memmove nie jest zoptymalizowany w ten sposób ze względu na możliwość nakładania się obszarów danych.

strlen-podobne konstrukcje kodu

Podobnie jak memcpy, funkcja biblioteki strlen jest często optymalizowana przez niektóre kompilatory do prostych instrukcji asemblera x86. Po raz kolejny oszczędza to narzut wprowadzony przez wywołanie funkcji. Dla tych, którzy nie są zaznajomieni z kodem generowanym przez kompilator, konstrukcja kodu strlen może początkowo wydawać się dziwna. Ogólnie wygląda to tak, jak poniżej:

```
mov edi, string

or ecx, 0xffffffff

xor eax, eax

repne scasb

not ecx

dec ecx
```

Wynikiem tych instrukcji jest to, że długość łańcucha jest przechowywana w rejestrze ECX. Instrukcja repne scasb skanuje z EDI znak przechowywany w młodszym bajcie EAX, który w tym przypadku jest zerem. Dla każdego znaku, który sprawdza ta operacja, dekrementuje ECX i zwiększa EDI. Pod koniec operacji repne scasb, gdy zostanie znaleziony bajt null, EDI wskazuje jeden znak za bajtem null, a ECX to ujemna długość łańcucha minus dwa. Logiczne NIE ECX, po którym następuje dekrementacja, skutkuje poprawną długością ciągu w ECX. Często zdarza się, że pod edi, ecx następuje bezpośrednio po instrukcji not ecx, która przywraca EDI do pierwotnej pozycji. Ta konstrukcja kodu będzie szeroko stosowana w każdym kodzie, który obsługuje dane ciągów; dlatego powinieneś to rozpoznać i zrozumieć, jak to działa.

Konstrukcje kodu C++

Większość nowoczesnego zamkniętego kodu źródłowego w systemach operacyjnych i serwerach jest napisana w C++. Pod wieloma względami ten kod jest skonstruowany w sposób bardzo podobny do zwykłego kodu C. Konwencje wywoływania są bardzo zbliżone, a w przypadku kompilatorów obsługujących zarówno C, jak i C++ używane są te same aparaty generowania kodu zestawu. Jednak pod pewnymi względami inspekcja kodu C++ jest inna i należy wspomnieć o kilku specjalnych przypadkach. Ogólnie rzecz biorąc, audyt plików binarnych składających się z kodu C++ jest nieco trudniejszy niż ten napisany w zwykłym C; jednak po pewnym zapoznaniu się nie jest to duży skok.

Wskaźnik this

Wskaźnik this odnosi się do konkretnej instancji klasy, do której należy bieżąca funkcja (metoda). musi to zostać przekazane do funkcji przez wywołującego; jednak nie jest przekazywany, jak może być zwykły argument funkcji. Zamiast tego wskaźnik this jest przekazywany w rejestrze ECX. Ta konwencja wywoływania używana w kodzie C++ nosi nazwę thiscall. Poniższy kod przedstawia przykład funkcji przekazującej wskaźnik klasy do innej funkcji:

```
push edi

push esi

push [ebp+arg_0]

lea ecx, [ebx+5Ch]

call ?ParseInput@HTTP_HEADERS@@QAEHPBDKPAK@Z
```

Jak widać, wskaźnik jest przechowywany w rejestrze ECX bezpośrednio przed wywołaniem funkcji. W tym przypadku wartość przechowywana w ECX jest wskaźnikiem do obiektu HTTP_HEADERS. Ponieważ rejestr ECX jest dość ulotny, wskaźnik ten jest często przechowywany w innym rejestrze po wywołaniu funkcji, ale prawie zawsze jest przekazywany w rejestrze ECX.

Rekonstrukcja definicji klas

Przy analizie kodu C++ bardzo pomocne jest zrozumienie struktury obiektu. Zebranie tych informacji może być trudne, jeśli audytor nie szuka we właściwych miejscach; wiele struktur obiektów jest dość złożonych i zawiera obiekty zagnieżdżone. Ogólna metodologia rekonstrukcji obiektów polega na znalezieniu wszystkich dostępów względem wskaźnika obiektu, a zatem wyliczeniu członków tej klasy. W większości przypadków typy tych elementów członkowskich muszą być wywnioskowane lub odgadnięte, ale w niektórych przypadkach można określić, czy są one używane jako argumenty znanych funkcji, czy w innym znajomym kontekście. Jeśli próbujesz zrekonstruować obiekt ręcznie, zazwyczaj najlepszym miejscem do rozpoczęcia poszukiwań jest konstruktor i dekonstruktor dla tej klasy. Są to funkcje, które inicjują i zwalniają obiekty, a zatem są funkcjami, które generalnie uzyskują dostęp do większości elementów obiektu w dowolnym miejscu. Te dwie funkcje zazwyczaj dostarczają wielu informacji o klasie, ale niekoniecznie wszystkich niezbędnych. Może być konieczne przejście do innych metod w klasie, aby uzyskać pełniejszy obraz obiektu. Jeśli program zawiera informacje o symbolach, konstruktor i destruktor można szybko znaleźć dla dowolnej aplikacji. Będą one miały notację Classname::Classname i Classname::~Classname. Jeśli jednak nie można ich znaleźć po nazwie, często można je rozpoznać po ich strukturze i miejscu odniesienia. Konstruktory są często liniowymi blokami kodu, które inicjują dużą liczbę elementów struktury. Są one na ogół pozbawione porównań lub skoków warunkowych i często powodują wyzerowanie elementów konstrukcyjnych lub dużych części konstrukcji. Destruktory często uwalniają wielu członków struktury. Halvar Flake napisał doskonałe narzędzie (OBJRec) jako wtyczkę do IDA Pro co zabierze trochę nudy z ręcznego wyliczania elementów struktury dla obiektu.

vtables

Sposób, w jaki kompilatory radzą sobie z wirtualnymi tabelami funkcji (vtables) może bardzo irytować podczas audytu plików binarnych. Gdy wywoływana jest funkcja z tabeli vtable, wyśledzenie dokładnie, dokąd zmierza to wywołanie, może być trudne bez analizy w czasie wykonywania. Na przykład następujący typ kodu będzie często spotykany w skompilowanym kodzie C++:

```
mov eax, [esi]

lea ecx, [ebp+var_8]

push ebx

push ecx

mov ecx, esi

push [ebp+arg_0]

push [ebp+var_4]

call dword ptr [eax+24h]
```

W tym przypadku ESI zawiera wskaźnik do obiektu, a niektóre wskaźniki funkcji są wywoływane z jego vtable. Aby odkryć, dokąd prowadzi to wywołanie funkcji, konieczne jest poznanie struktury vtable. Strukturę zazwyczaj można znaleźć w konstruktorze klasy. W takim przypadku w konstruktorze

znajduje się następujący kod: mov dword ptr [esi], offset vtable. W tym konkretnym przykładzie łatwo było zlokalizować wywołanie funkcji w vtable, ale dość często wywoływany jest wskaźnik funkcji z vtable obiektu zagnieżdżonego. Rozwiązanie tego typu sytuacji może wymagać sporo pracy.

Szybkie, ale przydatne ciekawostki

Niektóre z przedstawionych tu punktów są dość oczywiste, ale mimo to przydatne; jeśli jeszcze ich nie znasz, przegapisz kilka kluczowych punktów podczas audytu plików binarnych.

■■ Wartość zwracana dla wywołania funkcji znajduje się w rejestrze EAX.

■■ Instrukcje skoku JL/JG są używane w porównaniach ze znakiem.

■■ Instrukcje skoku JB/JA są używane w porównaniach bez znaku.

■■ Znak MOVSX rozszerza rejestr docelowy, podczas gdy MOVZX rozszerza go o zero.

Ręczna analiza binarna

Sprawdzona w czasie ręczna metoda odczytywania deasemblacji jest nadal bardzo skutecznym sposobem lokalizowania luk w plikach binarnych. W zależności od jakości kodu audytor może stosować różne podejścia podczas rozpoczynania ręcznego audytu binarnego aplikacji.

Szybkie badanie połączeń bibliotecznych

Jeśli jakość kodu jest dość zła, zwykle warto zacząć od szukania prostych błędów w kodowaniu, które w dzisiejszych czasach można znaleźć tylko w oprogramowaniu o zamkniętym kodzie źródłowym. Proste wywołania funkcji tradycyjnej biblioteki problemów powinny być badane z nadzieją na szybkie znalezienie błędu. Należy zbadać tradycyjne funkcje problemowe, takie jak strcpy, strcat, sprintf i wszystkie ich pochodne. System operacyjny Windows ma wiele wariantów tych funkcji, w tym wersje dla zestawów znaków szerokich i ASCII. Na przykład funkcje podobne do strcpy mogą obejmować strcpy, lstrcpyA, lstrcpyW, wcscpy i funkcje niestandardowe zawarte w aplikacji. Innym częstym źródłem problemów w systemie Windows jest funkcja MultiByteToWideChar. Szóstym argumentem funkcji jest rozmiar bufora docelowego szerokich znaków. Jednak rozmiar jest określany jako liczba szerokich znaków, a nie całkowity rozmiar bufora. Powszechnym błędem kodowania było historycznie przekazywanie wartości sizeof() jako szóstego argumentu, a ponieważ każdy szeroki znak ma 2 bajty, może to potencjalnie prowadzić do dwukrotnego zapisu rozmiaru bufora docelowego. Doprowadziło to w przeszłości do luk w zabezpieczeniach serwera internetowego IIS firmy Microsoft.

Podejrzane pętle i instrukcje zapisu

Kiedy proste wywołania API nie wykryją oczywistych luk w zabezpieczeniach, czas na prawdziwy audyt binarny. Podobnie jak inspekcja w każdej innej formie, wiąże się to z uzyskaniem pewnego zrozumienia badanej aplikacji i przeczytaniem odpowiednich sekcji kodu. Jeśli aplikacja ma oczywisty punkt wyjścia do inspekcji, na przykład procedurę przetwarzającą niezaufane dane zdefiniowane przez atakującego, zacznij tam i czytaj dalej. Jeśli żaden taki punkt nie jest oczywisty, często można znaleźć dobry punkt wyjścia, szukając w kodzie informacji specyficznych dla protokołu. Na przykład serwer sieci Web analizujący żądania przychodzące najprawdopodobniej rozpocznie się od analizy metody żądania; przeszukiwanie pliku binarnego pod kątem typowych metod żądań może być dobrym sposobem na znalezienie punktu wyjścia do audytu. Niektóre typowe konstrukcje kodu wskazują na niebezpieczny kod, który może zawierać przepełnienia bufora. Oto kilka przykładów. Zmienna indeksowana zapisuje do tablicy znaków:

mov [ecx+edx], al

Zmienna indeksowana zapisuje do lokalnego bufora stosu:

mov [ebp+ecx-100h], al.

Zapis do wskaźnika, po którym następuje przyrost tego wskaźnika:

mov [edx], ax

inc edx

inc edx

Rozszerzona kopia znaku z bufora kontrolowanego przez atakującego:

mov cl, [edx]

movsx eax, cl

Dodawanie lub odejmowanie od rejestru zawierającego dane kontrolowane przez atakującego (prowadzące do przepełnienia liczby całkowitej):

mov eax, [edi]

add eax, 2

cp eax, 256

jae error

Obcięcie wartości w wyniku zapisania jako 16- lub 8-bitowej liczby całkowitej:

push edi

call strlen

add esp, 4

mov word ptr [ebp-4], ax

Rozpoznawanie tych konstrukcji kodu i wielu im podobnych powinno umożliwić zlokalizowanie szerokiego zakresu luk w zabezpieczeniach pamięci w plikach binarnych.

Zrozumienie wyższego poziomu i błędy logiczne Chociaż większość wykrytych dzisiaj luk dotyczy uszkodzenia pamięci, niektóre błędy są całkowicie niezwiązane z uszkodzeniem pamięci i są po prostu błędami logicznymi w aplikacji. Dobrym tego przykładem jest odkryta kilka lat temu luka podwójnego dekodowania IIS. Tego typu luki są wprawdzie trudne do wykrycia za pomocą analizy binarnej i wymagają albo szczęścia, albo bardzo dobrego zrozumienia aplikacji do zlokalizowania. Oczywiście nie ma konkretnego sposobu na znalezienie tego typu błędów, ale ogólnie najlepszym sposobem na znalezienie tego typu problemów jest dogłębne zbadanie kodu, który uzyskuje dostęp do krytycznych zasobów na podstawie danych określonych przez użytkownika. Pomaga mieć kreatywność i otwarty umysł podczas szukania tego typu błędów, ale dużo wolnego czasu jest oczywistym wymogiem.

Graficzna analiza plików binarnych

Niektóre funkcje, szczególnie te, które są bardzo duże lub złożone, mają większy sens, gdy są wyświetlane graficznie. Na wykresie pewne złożone pętle stają się łatwo rozpoznawalne; znacznie trudniej jest pomylić sekcje kodu, gdy oglądasz je jako część dużego wykresu, a nie jako liniowy deasembler. IDA Pro może generować wykresy dowolnej funkcji, z każdym węzłem będącym ciągłą

sekcją kodu. Węzły są połączone gałęziami lub przepływem wykonywania, a każdy węzeł ma prawie gwarancję, że zostanie wykonany jako ciągły blok kodu. Większość wykresów jest zbyt dużych, aby można je było wygodnie oglądać jako całość na większości monitorów. W związku z tym bardzo przydatne jest drukowanie wydruków wykresów funkcji, które często obejmują wiele stron, a następnie analizowanie ich na papierze. Jednak silnik graficzny programu IDA Pro błędnie zinterpretuje kod wygenerowany przez kompilator. Na przykład nie będzie zawierał fragmentów kodu generowanych przez MSVC++ w wykresie funkcji, co prowadzi do niekompletnych i często bezużytecznych wykresów. Wtyczka graficzna dla IDA Pro stworzona przez Halvara Flake'a będzie poprawnie zawierała te fragmenty kodu, tworząc kompletne i użyteczne wykresy dla kodu skompilowanego MSVC++.

Dekompilacja ręczna

Niektóre funkcje są zbyt duże, aby można je było poprawnie przeanalizować podczas demontażu. Inne zawierają bardzo złożone konstrukcje pętli, których bezpieczeństwa nie można łatwo określić za pomocą tradycyjnej analizy binarnej. Alternatywą, która może działać w takich przypadkach, jest ręczna dekompilacja. Dokładna dekompilacja będzie oczywiście łatwiejsza do audytu niż deasemblacja, ale należy dołożyć wszelkich starań, aby każda wykonana praca była dokładna. Nie ma sensu przeprowadzać audytu dekompilacji z błędami. Pomocne jest całkowite odłożenie na bok nastawienia na audyt bezpieczeństwa (jeśli to możliwe) i po prostu stworzenie reprezentacji kodu źródłowego funkcji. W ten sposób dekompilacja jest mniej skażona myśleniem życzeniowym.

Przykłady luk w zabezpieczeniach plików binarnych

Spójrzmy na konkretny przykład zastosowania analizy binarnej do wyszukiwania luki w zabezpieczeniach.

Błędy Microsoft SQL Server

Dwóch współautorów tej książki, David Litchfield i Dave Aitel, odkryli kilka bardzo poważnych luk w Microsoft SQL Server. Błędy SQL Server zostały wykorzystane w takich robakach jak Slammer i miały daleko idące konsekwencje dla bezpieczeństwa sieci. Szybkie badanie podstawowych usług sieciowych w niezałatanej bibliotece sieciowej SQL Server szybko ujawnia źródło tych błędów. Luka wykryta przez Litchfield jest wynikiem niesprawdzonego wywołania sprintf w procedurach przetwarzania pakietów usługi rozwiązywania problemów SQL.

```
mov edx, [ebp+var_24C8]

push edx

push offset aSoftwareMic_17 ; “SOFTWARE\\Microsoft\\Microsoft SQL

Server”...

push offset aSSMssqlserverC ; “%s%s\\MSSQLServer\\CurrentVersion”

lea eax, [ebp+var_84]

push eax

call ds:sprintf

add esp, 10h
```

W tym przypadku var_24C8 zawiera dane pakietowe odczytywane z sieci i może mieć rozmiar zbliżony do 1024 bajtów, a var_84 jest 128-bajtowym lokalnym buforem stosu. Konsekwencje tej operacji są oczywiste i niezwykle oczywiste podczas badania binarnego. Luka SQL Server Hello wykryta przez Dave'a Aitela jest również wynikiem niesprawdzonej operacji na łańcuchu, w tym przypadku po prostu strcpy.

```
mov eax, [ebp+arg_4]

add eax, [ebp+var_218]

push eax

lea ecx, [ebp+var_214]

push ecx

call strcpy

add esp, 8
```

Bufor docelowy, var_214, to 512-bajtowy lokalny bufor stosu, a łańcuch źródłowy to po prostu dane pakietowe. Ponownie, dość uproszczone błędy mają tendencję do utrzymywania się dłużej w oprogramowaniu o zamkniętym kodzie źródłowym, które jest powszechnie dostępne tylko jako plik binarny.

Luka w zabezpieczeniach LSD RPC-DCOM

Niesławna i szeroko wykorzystywana luka w zabezpieczeniach wykryta przez The Last Stage of Delirium (LSD) w interfejsach RPC-DCOM była wynikiem niesprawdzonej pętli kopiowania ciągów podczas analizowania nazw serwerów z nazw ścieżek UNC. Ponownie, gdy znajduje się w rpcss.dll, pętla kopiowania pamięci jest dość oczywistym zagrożeniem bezpieczeństwa.

```
mov ax, [eax+4]

cmp ax, '\'

jz short loc_761AE698

sub edx, ecx

loc_761AE689:

mov [ecx], ax

inc ecx

inc ecx

mov ax, [ecx+edx]

cmp ax, '\'

jnz short loc_761AE689
```

Nazwa ścieżki UNC ma format \\serwer\udział\ścieżka i jest przesyłana jako szeroki ciąg znaków. Pętla kopiowania w powyższym kodzie pomija pierwsze 4 bajty (dwa znaki odwrotnego ukośnika) i kopiuje do bufora docelowego, dopóki nie zostanie wyświetlony kończący ukośnik odwrotny, bez sprawdzania granic. Konstrukcje pętli, takie jak ta, są dość często źródłem luk w zabezpieczeniach pamięci.

Luka w zabezpieczeniach IIS WebDAV

Luka IIS WebDAV ujawniona w biuletynie Microsoft Security Bulletin MS03-007 była dość rzadkim przypadkiem, w którym exploit 0day został odkryty na wolności i skutkował poprawką bezpieczeństwa. Ta luka nie została wykryta przez badaczy bezpieczeństwa, ale przez stronę trzecią o złośliwych zamiarach. Rzeczywista luka w zabezpieczeniach wynikała z zawinięcia 16-bitowych liczb całkowitych, które często występuje w podstawowych funkcjach ciągu biblioteki wykonawczej systemu Windows. Typy przechowywania danych używane przez funkcje, takie jak RtlInitUnicodeString i RtlInitAnsiString, mają pole długości, które jest 16-bitową wartością bez znaku. Jeśli do tych funkcji zostaną przekazane ciągi o długości przekraczającej 65 535 znaków, pole długości zostanie zawinięte i w wyniku powstanie ciąg, który wydaje się być bardzo mały. Luka w zabezpieczeniach IIS WebDAV była wynikiem przekazania ciągu dłuższego niż 64 KB do RtlDosPathNameToNtPathName_U, co spowodowało zawinięcie pola długości ciągu Unicode i bardzo dużego ciągu z polem o małej długości. Ten konkretny błąd jest dość subtelny i najprawdopodobniej nie został wykryty podczas audytu plików binarnych; jednak wraz z praktyką i czasem można znaleźć tego typu problemy. Podstawowa struktura danych dla ciągu Unicode lub ANSI wygląda następująco:

```
typedef struct UNICODE_STRING {

unsigned short length;

unsigned short maximum_length;

wchar *data;

};
```

Kod w RtlInitUnicodeString wygląda następująco:

```
mov edi, [esp+arg_4]

mov edx, [esp+arg_0]

mov dword ptr [edx], 0

mov [edx+4], edi

or edi, edi

jz short loc_77F83C98

or ecx, 0FFFFFFFFh

xor eax, eax

repne scasw

not ecx

shl ecx, 1

mov [edx+2], cx // possible truncation here

dec ecx

dec ecx

mov [edx], cx // possible truncation here
```

W tym przypadku długość szerokiego ciągu jest określana przez repne scsw i mnożona przez dwa, a wynik jest przechowywany w 16-bitowym polu struktury. W funkcji wywoływanej przez RtlDosPathNameToNtPathName_U widoczny jest następujący kod:

```
mov dx, [ebp+var_30]

movzx esi, dx

mov eax, [ebp+var_28]

lea ecx, [eax+esi]

mov [ebp+var_5C], ecx

cmp ecx, [ebp+arg_4]

jnb loc_77F8E771
```

W tym przypadku var_28 to kolejna długość ciągu, a var_30 to długa struktura UNICODE_STRING atakującego z obciętą 16-bitową wartością długości. Jeśli suma długości dwóch ciągów jest mniejsza niż arg_4, co jest długością docelowego bufora stosu, dwa ciągi są kopiowane do bufora docelowego. Ponieważ jeden z tych ciągów jest znacznie większy niż zarezerwowane miejsce na stosie, występuje przepełnienie. Pętla kopiowania znaków jest dość standardowa i łatwo rozpoznawalna. Wygląda to tak:

```
mov [ecx], dx

add ecx, ebx

mov [ebp+var_34], ecx

add [ebp+var_60], ebx

loc_77F8AE6E:

mov edx, [ebp+var_60]

mov dx, [edx]

test dx, dx

jz short loc_77F8AE42

cmp dx, ax

jz short loc_77F8AE42

cmp dx, '/'

jz short loc_77F8AE42

cmp dx, '.'

jnz short loc_77F8AE63

jmp loc_77F8B27F
```

W takim przypadku ciąg jest kopiowany do bufora docelowego do momentu napotkania kropki (.), ukośnika (/) lub bajtu null. Chociaż ta konkretna luka spowodowała zapisanie się na szczyt stosu i

awarię wątku, wskaźnik obsługi wyjątków SEH został nadpisany, co spowodowało wykonanie dowolnego kodu.

Podsumowanie

Wiele luk wykrytych w produktach o zamkniętym kodzie źródłowym to te, które zostały wyeliminowane z oprogramowania o otwartym kodzie źródłowym wiele lat temu. Z powodu niektórych wyzwań związanych z audytem plików binarnych, większość tego oprogramowania jest niedostatecznie zbadana lub przetestowana tylko w sposób rozmyty, a wiele luk wciąż czai się niezauważonych. Chociaż audyt binarny wiąże się z pewnym nakładem pracy, nie jest on dużo trudniejszy niż audyt kodu źródłowego i po prostu wymaga trochę więcej czasu. W miarę upływu czasu wiele bardziej oczywistych luk zostanie przetestowanych w oprogramowaniu komercyjnym, a aby znaleźć bardziej subtelne błędy, audytor będzie musiał przeprowadzić bardziej dogłębną analizę binarną. Audyty binarne mogą w końcu stać się tak powszechne, jak przeglądanie kodu źródłowego - jest to z pewnością dziedzina, w której wciąż pozostaje wiele do zrobienia.