Wprowadzenie do Reverse Engineeringu

Co to jest reverse engineering?

Reverse engineering będziemy traktować po prostu jako rozważanie jak oprogramowanie, do którego nie masz kodu źródłowego, do jego cech i funkcji, możesz albo zmodyfikować ten kod, lub odtworzyć w inny niezależny sposób. Generalnie, podstawy reverse engineeringu są trudne. Jednak, znajdziemy takie narzędzia dostępne dla nas, poprowadzimy dobre notatki z tego co się będzie działo, powinniśmy móc wyodrębnić te informacje jakich potrzebujemy dla naszych spraw: tworzenie modyfikacji i hakowanie uzyskanego oprogramowania, do którego nie mamy kodu źródłowego dla wykonania tego, a co nie było pierwotnie skierowane dla nas Dlaczego reverse engineering? Każdy entuzjasta komputerów (a zasadniczo każdy entuzjasta generalnie) jest osobą która uwielbia rządzić. Kochamy szczegóły. Kochamy odkrywać rzeczy. Jeśli mamy kod źródłowy do oprogramowania, wszystko jest fajne. Ale niestety, nie zawsze jest tak miło. Co więcej, oprogramowanie do którego nie masz kodu źródłowego jest zazwyczaj najbardziej interesującym rodzajem oprogramowania. Czasami może być ciekawe jak pracuje określona funkcja zabezpieczenia, lub czy ochrona kopii jest rzeczywiście niełamliwa, i czasami chcemy wiedzieć jak określona funkcja jest implementowana.

I. PROCES KOMPILACJII

Kompilacja generalnie jest podzielona z grubsza na 5 etapów: Przetwarzanie wstępne, Parsowanie , Tłumaczenie, Assemblacja i Linkowanie. Wszystkie 5 etapów jest zaimplementowanych w jednym programie w UNIX, nazwanym cc, lub w naszym przypadku, bcc (lub g++) Ogólny porządek rzeczy idzie tak gcc -> gcc -E -> gcc -S -> as -> ld Pod Windows, jednak, proces jest trochę bardziej zamaskowany, ale kiedy zagłębimy się w MSVC+ +, w zasadzie jest to to samo. Zwróć uwagę ,że GNU toolchain jest dostępne pod Windows, chociaż zarówno projekt MinGW jak też Cygwin Project zachowują się tak samo pod UNIX. Cygwin dostarcza całej kompatybilności warstwy POSIX i środowiska UNIX'owego, gdzie podobnie jak MinGW dostarcza samego GNU buildchain,i pozwala na zbudowanie natywnych aplikacji okienkowych bez konieczności dostarczania dodatkowycg dll'i

KOMPIlATOR

Pomimo ich pozornie różnych podejść do środowiska projektowego, zarówno UNIX jak i Windows dzielą wspólną specjalizowaną architekturę kiedy chodzi o kompilatory. Generowanie plików wykonywalnych jest zasadniczo obsługiwane w całym zakresie na obu systemach przez jedne program : kompilator. Oba systemy mają pojedynczy fronton wykonywalny, który działa jako klej dla wszystkich 5 kroków wspomnianych powyżej.

PREPROCESOR C

Preprocesor obsługuje logikę poza dyrektywami # w C. Uruchamia się w pojedynczym przebiegu i zazwyczaj zastępczym silnikiem.

gcc -E

gcc -E uruchamia tylko etap preproceosra. Umieszcza wszystkie pliki include w pliku .c, jak również tłumaczy wszystkie makra do kodu inline C. Możesz dodać plik -o dla przekierowania pliku

cl -E

Podobnie cl -E, również uruchamia tylko etap preprocesora, wysyłając wyniki do standardowego wyjścia

ETAPY PARSOWANIA I TŁUMACZENIA

Etapy parsowania i tłumaczenia są najbardziej użytecznymi etapami kompilatora. Później, użyjemy tych funkcjonalności dla nauczenia się asemblera, i poczuć sposób pisania kodu generowanego przez kompilator pod pewnym warunkami. Niestety, świat UNIX'a i świat Windows różnią się w swoich wyborach składni asemblera. Trzeba mieć nadzieję ,że odsłonięcie obu tych metod składniowych zwiększy się elastyczność odczytu , kiedy przechodzimy między tymi dwoma środowiskami. Zwróć uwagę ,że większość narzędzi GNU pozwala na elastyczne wybory składni Intela, powinniśmy sobie życzyć sobie aby wybrać

składnię i jej się trzymać.
gcc -S
gcc -S pobiera pliki .c jako dane wejściowe i podaje dane wyjściowe jako pliki asemblacji .s w składni AT&T. Jeśli życzysz sobie składni Intel, dodaj opcję -masm=intel. Dal osiągnięcia powiązań między zmiennymi a użyciem stosu, użyjemy -fverbose-asm do flag. gcc może być wywoływane z różnymi opcjami optymalizacji, które mogą robić ciekawe rzeczy z wyjściowym kodem asemblerowym. Istniej między 4 a 7 ogólnymi klasami optymalizacyjnymi, które mogą być określone z -0N, gdzie 0 <= N <= 6. 0 oznacza brak optymalizacji (domyślnie) a 6 oznacza maksimum, chociaż często żadna optymalizacja nie jest wykonywana po 4, zależnie od architekturyi wersji gcc. Jest również kilka szczegółowych opcji, które są określone flagą -f. Najbardziej interesującymi są -funroll-loops, -finline-functions i -fomit-frame-pointer. Pętla rozwijana oznacza rozwinięcie pętli tak ,że jest n kopii kodu dla n iteracji pętli (tj. żadnych instrukcji jmp na szczycie pętli) W nowoczesnych procesorach, ta optymalizacja jest nieistotna. Funkcje inline oznaczają w rzeczywistości konwersję wszystkich funkcji w pliku do makr, i umieszcza kopie ich kodu bezpośrednio w linii w funkcji wywoływanej (słowo kluczowe inline w C++). Stosuje się to tylko do funkcji wywołanych w ty samym pliku C do ich definicje. Jest to również relatywnie mała optymalizacja. Pominięcie wskaźnika ramki (wskaźnik bazowy) zwalnia dodatkowy rejestr , do użycia w programie. Jeśli masz więcej niż 4 mocno używane zmienne lokalne , może to być dużą zaletą, w przeciwnym razie jest to tylko niedogodność (i czyni debugging dużo trudniejszym)
cl -S
Podobnie, cl.exe ma opcję -S, która wygeneruje kod asemblerowy, i ma również kilka opcji optymalizacji. Niestety, cl nie wydaj zezwolenia optymalizacji kontrolowanej podobnie jak z poziomu gcc. Główną opcja optymalizacji jaką oferuje cl są predefiniowane dla szybkości lub przestrzeni. Parę opcji, które są podobne do tych co oferuje gcc to:
☐ Ob - funckja inlien (- finline-functions)
☐ Oy ☐ włącza pomijanie wskaźnika ramki (-fomit-frame-pointer)

ETAP ASSEMBLACJI

Etap assemblacji jest wtedy ,kiedy kod asemblerowy jest tłumaczony prawie bezpośrednio do instrukcji maszynowej. Mogą również wystąpić minimalny preprocesing, wypełnienie i przeporządkowanie instrukcji. Nie będziemy się koncentrować na tym zbytnio, ponieważ staje się to widoczne podczas disasemblacji.
GNU as
as jest asemblerem GNU. Pobiera dane wejściowe w składni AT&T i Intela jako plik asm i generuje plik wynikowy .o
MASM
MASM jest asemblerem Microsoft.

ETAP ŁĄCZENIA

Zarówno Windows jak i UNIX mają podobne procedury linkowania, chociaż obsługa ich jest trochę inna. Oba systemy obsługują 3 style linkowania, i oba implementują to w wielce podobne sposoby.

Łączenie statyczne

Łączenie statyczne oznacza ,że dla każdej funkcji jaką wywołuje program, kod asembler do tej funkcji jest zawarty w pliku wykonywalnym. Wywołane funkcji wykonywane jest przez wywołanie adresu tego kodu bezpośrednio, w ten sam sposób w jaki są wywoływane funkcje programu

Łączenie dynamiczne

Łączenie dynamiczne oznacza ,że biblioteki istnieją w tylko jednym położeniu w całym systemie, a system pamięci wirtualnej systemu operacyjnego będzie odwzorowywał to pojedyncze położenie w przestrzeni adresowej programu , kiedy program jest ładowany. Adres pod którym występuje to odwzorowanie nie zawsze jest gwarantowany, chociaż

pozostaje stały kiedy jest budowany plik wykonywalny. Wywołanie funkcji jest wykonywane przez wykonanie wywołania do generowanej sekcji czasu kompilacji pliku wykonywalnego, wywołanej przez Procedure Linkage Table, PLT, lub tablicy skoku, która jest zasadniczo dużą instrukcji skoków do właściwych adresów w mapowanej pamięci.

Łączenie czasu przetwarzania

Łączenie czasu przetwarzania jest łączeniem, które ma miejsce kiedy program wymaga funkcji z biblioteki, która nie był łączona w czasie kompilacji. Biblioteka jest mapowana z dlopen() pod UNIX'em i LoadLibrary() pod Windows, obie zwracające uchwyt , który jest potem przekazywany do funkcji rozdzielczości symboli (dlsym() i GetProcAddress()), które w rzeczywistości zwracają wskaźnik funkcji, który może być wywołany bezpośrednio z programu jak gdyby były normalnymi funkcjami. To podejście jest często stosowane przez aplikacje do załadowania pluginów bibliotek określonych przez użytkowników z dobrze zdefiniowanymi funkcjami inicjalizacji. Takie funkcje inicjalizacji zazwyczaj zgłaszają dalsze adresy funkcji do programu, który je ładuje.

ld/collect2

ld jest linkerem GNU. Będzie generował poprawny plik wykonywalny. Jeśli łączysz ponownie biblioteki współdzielone, będziesz chciał w rzeczywistości użyć tego co wywołuje gcc, czym jest collect2.

link.exe

Jest to linker MSVC++. Zwykle, będziesz przekazywał mu opcje pośrednio przez opcję łączenia ☐ cl. Jednakże , możesz użyć go bezpośrednio do łączenia plików obiektowych i plików .dll razem w plik wykonywalny. Z tego powodu, Windows wymaga abyś miał plik .lib (lub .def) dodatkowo do .dll'i aby połączyć je razem ponownie. Plik .lib jest tylko używany na etapie przejściowym, ale położenie do niego musi być określony w opcji ☐ LIBPATH.

II. ZBIERANIE INFORMACJI

Szeroki system zbierania informacji o procesie W Windows i w Linuksie, kilka aplikacji będzie dawało różne ilości informacji o uruchomionych procesach.

/proc

System plików /proc Linuksa zawiera wszystkiego rodzaju interesujące informacje, gdzie biblioteki i inne sekcje kodu są odwzorowane, które i gdzie pliki i gniazda są otwarte. System plików /proc zawiera katalog dla każdego aktualnie uruchomionego procesu. Tak więc, jeśli uruchomisz proces, którego pid to 1337, możesz wpisać katalog /proc/1337/ aby odkryć prawie wszystko o tym aktualnie uruchomionym procesie. Możesz tylko zobaczyć informację o procesie dla procesów które są twoje własne. Pliki w tym katalogu zmienia się z każdym Systemem UNIX. Ciekawe w Linuksie są : cmdline ☐ lista parametrów wiersza poleceń przekazywanych do procesu, cwd ☐ link do bieżącego katalogu roboczego , environ ☐ lista zmiennych środowiskowych dla procesu , exe ☐ link do wykonywalnego procesu ,fd ☐ lista deskryptorów pliku będącego używanym przez proces, maps - listy komórek pamięci będących używanych przez ten proces. Może być to widoczne bezpośrednio z gdb , dla znajdowania różnych użytecznych rzeczy.

Sysinternals Proecess Explorer

Sysintrenals dostarcza wszechstronnych zbiorów programów narzędziowych. W tym przypadku, Process Eplorer jest funkcjonalnym odpowiednikiem /proc. Może pokazywać odwzorowane informacje dll, pod jakimi adresami są jakie funkcje, jak również właściwości procesu, które zawierają tabulatory środowiskowe, id bezpieczeństwa, jakie pliki i obiekty są otwarte, jakiego typu obiekty są obsługiwane itd. Pozwala nam również na modyfikowanie procesów do których masz dostęp w sposób, który nie jest dostępny w /proc. Możesz zamknąć uchwyty, zmienić uprawnienia, otworzyć okna debuggowania i zmienić priorytety procesu.

Uzyskiwanie informacji linkowania

Pierwszym krokiem w zrozumieniu jak działa program jest analiza jakie biblioteki są łączone. Może to nam pomóc bezpośrednio tworzyć przewidywania z jakim typem programu działamy

i tworzyć wyobrażenia na temat jego zachowania.

ldd

ldd jest podstawowym narzędziem, które pokazuje nam jakie biblioteki programu łączy, lub czy jest łączony statycznie . Podaje nam również adres pod jakimi te biblioteki są odwzorowywane do przestrzeni wykonywalnej programu, który może być przydatny dla wywoływania funkcji w danych wyjściowych disasemblacji.

depends

depends jest narzędziem, które pochodzi z Microsoft SDK , jak również MS Visual Studio. Będzie pokazywał nam trochę informacji o programie. Nie tylko będzie listował dll'e, ale będzie listował jakie funkcje w tych DLL'ach będą importowane (używane) przez bieżący plik wykonywalny i pod jakimi adresami rezydują.

Uzyskiwanie informacji o funkcji

Kolejnym krokiem w reverse engeneeringu jest możliwość rozróżniania funkcjonalnych bloków w programach. Niestety, to może być dowód ,że to trochę trudne jeśli nie mamy dość szczęścia posiadać informacji debuggowania.

nm

nm listuje lokalne i biblioteczne funkcje, zmienne globalne i ich adresy binarne. Jednak , nie działa na binariach.

Podgląd aktywności systemu plików

lsof

lsof jest programem, który listuje wszystkie otwarte pliki przez uruchomione procesy w systemie. Otwarty plik może być plikiem regularnym, katalogiem, specjalnym blokiem pliku, specjalnym plikiem znakowym, wykonujący odniesienie tekstu, biblioteką, strumieniem lub plikiem sieciowym (gniazdo Internetowe, plik NFS lub gniazdo domeny UNIX) , Ma wiele opcji ale w tym domyślnym trybie daje szeroką listę otwartych plików. lsof nie jest instalowany domyślnie na większości Linuksów /. UNIX'ów, więc musimy go zainstalować sami. W niektórych dystrybucjach lsof instalujemy w /usr/bin, który domyślnie nie jest ścieżką dostępu i musisz ją dodać. Oto krótkie wyjaśnienie kilku skrótów lsof użyty w jego danych wyjściowych:

cwd current working directory (bieżący katalog roboczy)
mem memory □ mapped file (plik mapy pamięci)
pd parent directory (katalog rodzicielski)
rtd root directory (katalog główny)
txt program text (code and data) (tekstprogramu (kod i dane)
CHR dla specjalnego pliku znakowego
sock dla gniazda nieznanej domeny
unix dla gniazda domeny UNIX
DIR dla katalogu
FIFO dla specjalnego pliku FIFO

Jest to bardzo poręczne narzędzie kiedy przychodzi do badania zachowania programu.lsof ujawnia mnóstwo informacji o tym co proces robi pod powierzchnią.

Sysinternals Filemon

Analogiczny do lsof w świecie okienkowym jest program Sysinternals Filemon. Może nie tylko pokazać otwarte pliki, ale również odczytać , zapisać i zapytać o status. Ponadto, możesz filtrować określony proces i typ operacji. Bardzo użyteczne narzędzie.

Sysinternals Regmon

Rejestr Windows jest kluczową częścią systemu, który zawiera wiele sekretów. Aby spróbować zrozumieć jak program działa, zdecydowanie powinieneś wiedzieć jak cel współdziała z rejestrem. Przechowuje on informacje konfiguracyjne, hasła i inne pożyteczne informacje. Regmon od Sysinternals pozwala ci monitorować wszystkie aktywne lub wybrane rejestry w czasie rzeczywistym. Koniecznie musisz to zrobić jeśli planujesz działania na celu w Windows. Podgląd otwartych połączeń sieciowych Jest jeden przypadek gdzie zarówno

Linux jak i Windows maja dokładnie taką samą nazwę dla narzędzia i wykonują dokładnie to samo zadanie.

netstat

netstat jest użytecznym narzędziem, które jest obecne we wszystkich nowoczesnych systemach operacyjnych. Jest używany do wyświetlania połączeń sieciowych, tablicy routingu, statystyki interfejsu i więcej. Jak netstat może być użyteczny? Powiedzmy ,że próbujemy reverse engineeringu programu, który używa jakichś połączeń sieciowych. Szybkie spojrzenie na to co wyświetla netstat może dać nam trop gdzie program się łączy a po małym śledztwie być może dlaczego łączy się z tym hostem. netstat nie tylko pokazuje połączenia TCP/IP ale również połączenia gniazd domeny UNIX'a, które są używane w międzyprocesowej komunikacji w wielu programach. Jest wiele informacji jakie wynikają z netstat. Ale jakie jest ich znaczenie? Dane wyjściowe są podzielone na dwie części ☐ połączenia Internet i gniazda domeny UNIX. Tu opiszę internetową część danych wyjściowych netstat. Pierwsza kolumna pokazuje używany protokół (tcp, udp, unix) w określonym połączeniu. Kolejki wysyłania i odbierania dla niego są wyświetlane w kolejnych dwóch kolumnach, po których występuje informacja identyfikująca połączenie ☐ host źródłowy i port, host przeznaczenia. Ostatnia kolumna pokazuje stan połączenia. Ponieważ jest kilka etapów w otwieraniu i zamykaniu połączeń TCP, pole to pokazuje czy połączenie jest USTANOWIONE lub jest w jakimś innym stanie. SYN_SENT, TIME_WAIT, LISTEN są najczęściej widzianymi. Aby zobaczyć pełną listę dostępnych stanów zajrzyj na stronach podręcznika dla netstat. W zależności od przekazywanych opcji do netstat, możliwe jest wyświetlenie więcej informacji. W szczególności interesujące jest opcja -p (niedostępna w systemach UNIX) Pokazuje ona nam program, który używa pokazanego połączenia, co może pomóc nam określić zachowanie naszego celu. Innym zastosowaniem tej opcji jest odnalezienie programów spyware, które mogą być zainstalowane w naszym systemie. Pokazywanie wszystkich połączeń sieciowych i wyszukiwanie nieznanych wejść jest bezcennym narzędziem w odkrywaniu programów , które są nieświadome ,że wysyłają informacje do sieci. Może być to połączone z opcją -a pokazującą wszystkie połączenia. Domyślnie nasłuchiwane gniazda nie są wyświetlane w netstat.. Użycie -a wymusza pokazanie wszystkiego -n pokazuje numeryczne adresy IP zamiast nazw hostów.

netstat -p as normal user

(Nie wszystkie procesy są identyfikowane, nasze info o procesie nie będzie pokazane, musisz być rootem aby zobaczyć wszystko.) Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address Foreign Address State
PID/Program name
tcp 0 0 slack.localnet:58705 egon:ssh ESTABLISHED - tcp 0 0 slack.localnet:58766 winston:www ESTABLISHED
5587/mozilla-bin

netstat -npa as root user

Połączenia Active Internet
Proto Recv-Q Send-Q Local Address Foreign Address State
PID/Program name
tcp 0 0 0.0.0.0:139 0.0.0.0:* LISTEN
390/smbd
tcp 0 0 0.0.0.0:6000 0.0.0.0:* LISTEN 737/X
tcp 0 0 0.0.0.0:22 0.0.0.0:* LISTEN
78/sshd
tcp 0 0 10.0.0.3:58705 128.174.252.100:22 ESTABLISHED

13761/ssh
tcp 0 0 10.0.0.3:51766 10.0.0.1:22 ESTABLISHED

897/ssh
tcp 0 0 10.0.0.3:51765 10.0.0.1:22 ESTABLISHED
896/ssh
tcp 0 0 10.0.0.3:38980 128.174.252.105:22 ESTABLISHED
8272/ssh
tcp 0 0 10.0.0.3:58510 128.174.5.39:22 ESTABLISHED
13716/ssh
Widać z tego ,że Mozilla ma ustanowione połączenia z winston przez HTTP (port www to 80) W drugim przypadku widzimy demona SMB, X-serwer i demona ssh nasłuchujących połączeń przychodzących.

Zbieranie danych sieciowych

Zbieranie danych sieciowych jest zazwyczaj robione przez program zwany snifferm. To co ten program robi, to nastawienie karty ethernetowej na tryb odbierania i zbiera wszystkie informacje jakie widzi. Co to jest tryb odbierania? Ethernet jest medium nadawczym. Wszystkie komputery nadają swoje komunikaty po kablach i każdy może zobaczyć te komunikaty. Każda karta sieciowa z interfejsem sieciowym (NIC), posiada niezmienny adres fizyczny zwany adresem MAC (Media Access Control), który jest używany w protokole Ethernet. Kiedy wysyłamy dane kablami, OS określa dane przeznaczenia i tylko NIC z adresem przeznaczenia MAC będzie na bieżąco przetwarzał te dane. Wszystkie pozostałe NIC'i nie będą brały pod uwagę danych płynących kablami. W trybie odbierania, karta odbiera wszystkie dane jakie widzi i wysyła je do OS'a. W tym przypadku możesz zobaczyć wszystkie dane, które przepływają w segmencie sieci lokalnej. Istnieje kilka popularnych programów do sniffowania, które różnią się interfejsem użytkownika i możliwościami, ale każdy z nich wykonuje tą samą pracę. Oto kilka dobrych narzędzi, których używamy na codzień:

- ethereal ☐ jeden z najlepszych snifferów. Ma graficzny interfejs zbudowany z biblioteki GTK. To nie tylko sniffer ale również analizator protokołów. Dzieli przechwycone na fragmenty, pokazując znaczenie każdego fragmentu (np. flagi TCP takie jak SYN lub ACK, lub nawet kerberos lub nagłówek NTLM). Ponadto, ma doskonały mechanizm filtrowania pakietów i może zapisywać przechwycony ruch sieciowy, które pasują do filtra dla późniejszej analizy. Jest dostępny zarówno dla Windowsa jak i Linuksa i wymaga (jak prawie każdy sniffer) bibliteki pcap.
- tcpdump ☐ jedne z pierwszych snifferów. Jest aplikacją konsolową, któa wyświetla informacje na monitorze .. Zaletą jest to ,że przychodzi domyślnie z każdą dystrybucją Linuksa. Windosowska wersja nosi nazwę WinDump
- ettercap ☐ również sniffer konsolowy. Używa biblioteki ncurses dostarczającej konsoli GUI. Posiada wbudowaną możliwość ARP poisoningu i obsługę wtyczek, które dają nam możliwości modyfikacji danych w locie. Czyni to go bardzo odpowiednim dla wszystkich rodzajów ataków Man-In-The-Middle. Ettercap nie jest wielkim snifferem, ale nic nie zabrania ci przed użyciem ARP poisoningu i funkcji wtyczek.

Teraz kiedy wiesz coto jest sniffer , czas nauczyć się jak używać podstawowej funkcjonalności jednego z nich, aby móc zbierać dane sieciowe. Powiedzmy ,że chcesz wiedzieć jak klient pocztowy uwierzytelnia się i pobiera wiadomości z serwera. Ponieważ protokół jakiego używa to POP3, powinniśmy poinformować ethreal (lub wybrany sniffer) aby przechwytywał ruch tylko na porcie 110 lub pochodzących z portu 110. Jeśli masz wiele maszyn sprawdzających pocztę w tym samym czasie w sieci z hubem, możesz chcieć ograniczyć dopasowywanie tylko do twojej maszyny i serwera z jakim się łączysz. Tu mamy przykład przechwyconego pakietu w ethrealu: Ethreal dzieli pakiety dla nas, pokazując co dana część danych oznacza. Na przykład, 1 pokazuje nam poziom informacji Ethernet, takie jak adres źródłowy i przeznaczenia MAC.Wyjaśniane są również znaczenia kadego bitu wartości flagi. Zanalizujmy nagłówek informacji TCP, powiedzmy ,że ... ... bitów jest ustawionych a reszta nie. Używając przechwytywania pakietó można śledzić przepływ

protokołu dla lepszego zrozumienia jak działa apliakcja, lub nawet spróbować reverse engineeringu samego protokołu jeśli jest nieznany.

III. OKREŚLANIE INTERESUJĄCYCH FUNKCJI

Oczywiście ,że kod źródłowego nie mamy możliwości na zrozumienie wszystkich sekcji w całym programie. Więc musimy użyć różnych metod i odgadywanek w celu zawężenia poszukiwań do paru kluczowych funkcji

Rekonstrukcja funkcji i kontrola informacji

Problem polega na tym, że najpierw musimy określić, jakie części kodu są faktycznie funkcjami.Może to być trudne bez symboli debugowania. Na szczęście istnieje kilka narzędzi, które czynią nasze życie łatwiejszym

objdump

Objdump jest najbardziej użyteczny dla zdisasemblowania programu z opcją -d. Brakujące symbole powodują że dane wyjściowe są trochę zagadkowe. Opcja -j jest używana do określenia segmentu disasemblacji. Najprawdopodobniej będziemy chcieli segmentu .text, w którym leży cały kod programu. Zwróć uwagę ,że komuny na lewo objdump zawierają liczby szesnastkowe. Jest to w istocie rzeczywisty adres w pamięci gdzie ta instrukcja jest umieszczona Jej wartość binarna jest podana w kolejnej kolumnie, po której występuje jej mnemonik. objdump -B poda nam listong wszystkich funkcji bibliotecznych tego wywołanego programu Steve Barker napisał sprytny skrypt, któy tworzy objdump dużo bardziej czytelne które w rzeczywistości nie zawierają symboli. Skrypt został poprawiony przez Nasko Oskova. Teraz tworzy 3 przebiegi przez dane wyjściowe. Pierwszy przebieg buduje tablicę symboli wywoływanych funkcji i skoków do lokacji. Drugi przebieg znajduje obszar między dwoma retami i wstawia je do tablicy symboli jaki funkcje "nieużywane". Trzeci przebieg wyświetla przyjemnie oetykietowane dane wyjściowe i wyświetla drzewo wywołanie funkcji:
Stosowanie:
./disasm /path/to/binary > binary.asminfo
Jest / będzie kilka opcji wiersza polecenia tego narzędzia. Teraz obsługiwany jest -graph. Generuje on plik wywoływany call_graph, który zawiera definicję , która może być użyta z programem dot do generowania wizualnej reprezentacji grafu połączeń. Notka: Nieużywana funkcja oznacza tylko ,ze funkcja ta nie była wywołana BEZPOŚREDNIO. Jest możliwe ,że funkcja została wywołana przez wskaźnik funkcji.

Rozważanie celu

Pierwszy krok to odkrywanie to czego szukamy i znajomość tego czego szukamy. Jaka funkcja jest "interesująca" jest całkowicie uzależnione od twojego punktu widzenia. Czy szukasz ochrony przed kopiowaniem? Podejrzewasz jak to jest zrobione. Kiedy pokazuje się to przy wykonaniu programu? Czy chcesz zrobić audyt bezpieczeństwa programu? Czy jest jakieś niechlujne użycie łańcucha? Która funkcja używa strcmp, sprintf, itd? Jak używać malloc? Czy istnieje możliwość niewłaściwej alokacji pamięci?

Znajdowanie kluczowych funkcji

Jeśli możemy zawęzić nasze poszukiwania do kilku funkcji, które są istotne dla naszego celu, nasze życie stanie się dużo łatwiejsze.

Znajdowanie main()

Bez względu na nasz cel, prawie zawsze pomocne jest wiedzieć gdzie leży main(). Niestety, kiedy symbole debugowania są usunięte, nie jest to zawsze łatwe.W Linuksie wykonywanie programu faktycznie rozpoczyna się w miejscu określonym przez symbol _start , który jest dostarczonych przez gcc w bibliotece crt0 . Wykonuje a potem kontynuuje do _libc _start _main(), która zwraca _init dla każdej biblioteki w przestrzeni programu. Każdy _init() potem wywołuje globalne konstruktory, jakie masz w tej określonej bibliotece. Globalne konstruktory mogą być tworzone przez stworzenie globalnej instancji klas C++ z konstruktorem, lub przez określenie _attribute _ ((construktor) po protoypie funkcji. Po tym, wykonywanie jest w końcu przenoszone do main. Najłatwiejszą techniką jest próba użycia naszego przyjaciela ltrace i gdb razem z naszymi danymi zdisasemblowanymi. Sprawdzamy

adres powrotu pierwszych kilku funkcji ltrace -i , i wzajemne odniesienia do naszych asemblowanych danych wyjściowych i drzewa wywołania funkcji, co powinno dać nam informacje gdzie jest main. Musimy spróbować sztuczki z programem do wczesnego wyjścia, lub wyświetlić komunikat błędów przed zrobi się zbyt głęboko w wywołaniu stosu. Istnieją inne techniki. Na przykład, możemy mieć plik .c LD_PRELOAD z konstruktorem funkcji w nim zawartym. Możemy potem ustawić punkt przerwania na funkcję libctak ,że wywołanie jest również w głównym wykonywanym programie, i finish i stepi dopóki nie będziemy mieli pewności ,że znaleźliśmy main. Nawet lepiej, możemy ustawić punkt przerwania na funkcji _libc _start _main (która jest funkcją libc, a zatem zawsze mamy dla niej symbol) i wykonuje tę samą technikę zakończenia i wykonać pracę krokową dopóki osiągniemy to co wygląda dla nas jak main. W najgorszym wypadku, nawet bez wskaźnika ramki, powinniśmy móc pobrać adres funkcji dsyć wczesnym etapie łańcuchu wykonania dla nas.

Znajdowanie innych ciekawych funkcji

Prawdopodobnie dobrym pomysłem jest tworzenie listy wszystkich funkcja wywołujących exit. Mogą być one użyteczne dla nas. Inne techniki śledzenia interesujących nas funkcji obejmują :

1. Sprawdzeni dla jakich wywołanych funkcji niejasne widgety konstrukcji GUI użyte w oknie dialogowym proszą o numer seryjny produktu
2. Sprawdzenie referencji łańcucha aby dowiedzieć się które funkcje odnoszą się do referencji łańcuch jakie nas interesują. Na przykład jeśli program podaj tekst "Już zarejestrowane", wiedząc jaka jest funkcja wyjściowa tego łańcucha, jest on pomocny przy znajdowaniu zabezpieczenia tego określonego używanego programu
3. Uruchomienie programu w gdb a potem trafienie kontroli C kiedy zaczyna wykonywanie jakiejś ciekawej operacji Użycie stepi N powinno spowolnić sprawę i pozwolić ci na dokładniejsze badanie. Czasami jest to jednak zbyt wolne. Znajdź powszechnie używaną funkcję, ustaw punkt przerwania a spróbuj wykonać cont N
4. Sprawdzenie jaka funkcja wywołuje funkcje w warstwie gniazda BSD

Nakreślenie przepływu programu

Nakreśl ścieżki dostępu wykonania na drzewo w main, szczególnie dla funkcji jakie cię interesują. Możesz użyć disasm.pl dla generowania wykresu połączenia z opcją ☐ graph. Użycie jej włącza skrypt do generowania pliku wywołującego call_graph. Zawiera definicję wykresu połączenia w formacie używanym przez popularne narzędzie graficzne zwane dot. Dostarczenie tej definicji pliku w dot daje ładny (prawdopodobnie bardzo duży) plik graficzny z wizualnym przedstawieniem wykresu połączenia. Zadziwiające.

IV. ZROZUMIENIE ASSEMBLERA

Ponieważ wszystkie dane wyjściowe tych narzędzi są w składni AT&T, ci z was którzy znają składnię Intel/MASM mają trochę powtórnej nauki. Język asembler jest krokiem bliżej zrozumienia sprzętu niż języki wysokopoziomowe takie jak C czy C++. Tak więc rozumiejąc asembler, zrozumiesz jak działa sprzęt. Zacznijmy od zbioru komórek pamięci znanych jako rejestry CPU

Rejestry

Rejestry są jak zmienne lokalne CPU, z wyjątkiem tego ,że jest ich stała liczba. Dla CPU x86, są tylko 4 główne rejestry dla obliczeń całkowitych :A,B,C i D. Każdy z tych 4 rejestrów mogą być dostępne na 4 różne sposoby: jako 32 bitowe wartości (%eax), 16 bitowe wartości (%ax) i jako niska i wysoka wartość 8 bitowa (%al, %ah). Jest pięć rejestrów, które są używane okazjonalnie ☐ a mianowicie SI, DI, SP i BP. SI i DI pochodzą z czasów DOS'a kiedy ludzie używali adresowania segmentowego 64KB, i jak się okazuje, być mogą używane jako całkowite jak normalne rejestry. SP i BP są dwoma specjalnym rejestrami używanymi do obsługi obszaru pamięci nazwanego stosem. Jest jeden, ostatni rejestr, wskaźnik instrukcji IP, którego nie możesz modyfikować bezpośrednio, ale jest zmieniany przez instrukcje jmp i call. Jego wartością jest adres kolejnej instrukcji do wykonania.

Stos

Co to jest stos?
Stos jest tym co nazywamy strukturą danych Last In, First Out lub LIFO. Myśl o nim jako stosie talerzy. Najnowszy talerz (ostatni) odłożony na szczycie stosu jest pierwszym jaki jest usuwany. Pozwala to nam na zarządzanie stosem tylko z jednym rejestrem, nazwanym rejestrem SP. Stos jest rejonem pamięci , który jest obecny podczas całego życia programu. Jest tam gdzie są przechowywane zmienne lokalne, a także w jaki sposób są przekazywane argumenty wywołania funkcji. We wszystkich nowoczesnych komputerach, stos rośnie w dół, to znaczy elementy są odkładane na nim, rejestr SP jest zmniejszany o rozmiar odkładanego elementu. Z naszej wcześniejszej analogii, jak gdyby stos talerzy wisiał u sufitu, nowe talerze są wstawiane u dołu, a cały stos jest tak jakby był łapany aby zatrzymać je przed zrzuceniem. Tym łapaczem będzie rejestr SP. Tak więc stos zaczyna się od wysokiego adresu pamięci i działa w dół do niższego adresu. Jest tak ponieważ inna sekcja pamięci nazwanej stertą dorasta, i przydatna aby mieć dwie z nich rosnąc w kierunku siebie aby wypełnić pojedynczą pustą dziurę w przestrzeni adresowej programu. Notka: Łatwo się pogubić kiedy działamy ze stosem. Pamiętaj ,że kiedy on rośnie w dół, zmienne są jeszcze adresowane sekwencyjnie w górę. Tak więc tablica znaków b[4] pod esp 80 będzie miała b[0] to 80, b[1] powyżej tego 81, b[2] pod 82 a b[3] pod 83, czyli gdzie wskaźnik stosu był przed odłożeniem. Kolejne odłożenie umieści wskaźnik stosu pod 76
Praca ze stosem
Są dwie instrukcje, które działają ze stosem bezpośrednio : push i pop. Każda pobiera rejestr albo wartość jako argument. Push umieszcza swój argument na stosie, a potem zmniejsza Sp o rozmiar swojego argumentu (4 dla pushl, 2 dla pushw i 1 dla pushb) .Pop kopiuje wartość ze szczytu stosu do swojego argumentu, potem zwiększa SP. Pusha i popa odkładają i zdejmują wszystkie rejestry jedną instrukcją. Z tego powodu rozważanej szybkości, wartość nie jest ruszana, tylko rejestr SP zmienia wskazania kolejnej komórki na stosie. SP zawsze wskazuje wartość na szczycie stosu a nie niepoprawną pamięć. Mogą być używane również wyrażenia arytmetyczne dla modyfikowania SP dla tworzenia przestrzeni bezpośredniej pracy stosu pamięci z innymi instrukcjami.
Jak gcc działa ze stosem
Tuż przed wywołaniem funkcji, jej argumenty są odkładane na stos w odwrotnej kolejności. Następnie instrukcja call odkłada adres kolejnej instrukcji (tj. wartość IP po call) na stos, a potem CPU zaczyna wykonywanie od adresu wywołania przez skopiowanie tej wartości do niewidocznego rejestru wskaźnika instrukcji (IP) Wywołana funkcja zaczyna się co jest znane prolog funkcji, który odkłada bieżący wskaźnik bazowy na stos, a potem kopiuje bieżący wskaźnik stosu do wskaźnika bazowego, potem odejmuje od SP dość miejsca dla przechowywania zmiennych lokalnych. Wskaźnik bazowy używany jest potem do odniesienia do zmiennych i parametrów podczas wykonywania funkcji, ponieważ jego wartość nie ma wpływu na odkładanie i zdejmowanie. Zatem, wszystkie parametry mają stały dodatni offset od BP, gdzie jako zmienne lokalne mają stały ujemny offset od BP. Na koniec wykonywania funkcji, wskaźnik bazowy jest kopiowany do wskaźnika stosu podczas ret, a adres powrotu jest zdejmowany ze stosu i umieszczany w niewidocznym rejestrze IP dla powrotu do funkcji wywołującej.
Uzupełnienie do dwóch
Co to jest?
Uzupełnienie do dwóch jest określonym sposobem na liczby całkowite ze znakiem i jest obecne w prawie wszystkich nowoczesnych komputerach. Jest tak ze względu na fakt ,że uzupełnienie do dwóch ma kilka zalet:
1. Stosuje się te same zasady dodatkowo, nie ma dodatkowej pracy koniecznej do obliczenia sumy ujemnych liczb całkowitych
2. Łatwo zanegować liczbę
3. Najbardziej znaczący bit mówi nam o znaku: 0 to plus 1 to minus
Zwróć uwagę ,że kiedy używamy wartości ze znakiem zakres liczb jaki może być

przedstawiany przez określoną liczbę bitów jest mniejsza niż zwykle. Zakres to -2n-1 do +2n-1-1

Konwersja

Jest kilka sposobów na konwersję liczby binarnej bez znaku do formy uzupełnienia do dwóch ze znakiem. Najbardziej intuicyjny i łatwy do zapamiętania jest uzupełnienie każdego bitu liczby i dodanie jeden, Zobaczmy jak jest przedstawiane -13, więc konwertujemy ją do postaci binarnej:
0000 1101
Potem odwracamy wszystkie bity
1111 0010
Teraz dodajemy jeden do niej
1111 0011
Tak więc 1111 0011 to -13 w uzupełnieniu do dwóch
Drugą metodą jest uzupełnienie wszystkich bitów w lewo z prawej strony o 1 bit, ale nie wliczamy go (ale nie bit najbardziej z prawej strony, np 0001 0100) Brzmi trochę zawile, ale jest łatwiej jest zrozumieć jak to jest zrobione. Wróćmy do przykładu z -13
0000 1101
^
Odwracamy bity od lewej do prawej strony
1111 0011
I to wszystko. Otrzymaliśmy liczbę bez drugiego kroku z dodawaniem jeden. Jest też trzecia metoda, odejmowania liczby od 2n .Oto jak działa
1000 0000
-
0000 1101

1111 0011
Mogą być inne sposoby robienia tego, ale nie musisz znać ich wszystkich. Aby skonwertować liczbę ujemną w uzupełnieniu do dwóch, zastosujemy dokładnie tą samą procedurę jak opisane aby wrócić do wartości dodatniej tej liczby.

Pod kątem reverse engineeringu

Teraz wiemy co to jest uzupełnienie do dwóch, więc przedstawię przykład procesu reverse engineeringu. Użyję jednego z narzędzi omówionych wcześniej, objdump i disasm.pl, i spojrzymy na polecenie binarne ls. Jeśli spojrzysz na function7 (która zaczyna się od adresu 80495a8) linie takie jak poniższa pojawią się często:
80495be: 83 c4 f8 add $0xfffffff8,%esp
Co ta instrukcja robi? Dodaje tylko pewne stałe do rejestru wskaźnika stosu (%esp). Są dwa sposoby w jaki możesz spojrzeć na tą stałą. Jest to albo duża liczba bez znaku, lub ujemna liczba uzupełnienia do dwóch. Liczba ujemna 0x00000008 lub po prostu -8 dziesiętnie. To co ta linie robią to zmniejszenie wskaźnika stosu o 8 (alokowanie więcej przestrzeni)

Porządek bitów

Dlaczego ta sekcja? Z jednego prostego powodu- różne platformy używają różnej kolejności bitów Są dwa różne porządki bitów □ little endian i big endian. Może ktoś wie co to jest kolejność bitów? Kolejność bitów odnosi się do fizycznego rozkładu danych w pamięci. Kiedy struktura danych lub typ danych jest przedstawiany przez więcej niż jeden bajt, kolejność bitów ma znaczenie. Na przykład, jeśli weźmiemy long (4 bajty), najmniej znaczący bajt to 0 a najbardziej znaczący to 3. Jeśli jesteśmy na maszynie z little endian, long będzie reprezentowany w pamięci jak to (cóż niektóre maszyny nie pozwalają na bajty adresowalne, ale zapomnij o tym): 0x040 0 0x041 1 0x042 2 0x043 3. Na maszynie z big endian rozkład będzie wyglądał tak: 0x040 3 0x041 2 0x042 1 0x043 0. Spójrzmy na przykład. Najłatwiejszy sposób aby zobaczyć różnice w rozkładaniu bajtów jest zobaczenia jak łańcuch jest przechowywany w pamięci w różnych architekturach. Oto przykład programu który to

demonstruje:
#include
int main() {
char* test = "this is a string";
printf("%s\n", test);
}
Po skompilowaniu go, mamy takie dane wyjściowe z dwóch różnych sposobów disasemblacji na mszynie z Solarisem : objdump
11850: 74 68 69 73 call d1a2be1c <_end+0xd1a0a394>
11854: 20 69 73 20 unknown
11858: 61 20 73 74 call 8482e628 <_end+0x8480cba0>
1185c: 72 69 6e 67 call c9a6d1f8 <_end+0xc9a4b770>
11860: 00 00 00 00 unimp 0
gdb
0x11850 <_IO_stdin_used+8>: 0x74686973 0x20697320 0x61207374
0x72696e67
Teraz spójrzmy na to jak sama pamięć jest zorganizowana i jak jest reprezentowany łańcuch:
Adres - Kod - Litera

0x11850 - 74 - t
0x11851 - 68 - h
0x11852 - 69 - i
0x11853 - 73 - s
0x11854 -20 -
0x11855 - 69 - i
0x11856 -73 - s
0x11857 - 20 -
0x11858 - 61 - a
0x11859 - 20 -
0x1185a - 73 - s
0x1185b - 74 - t
0x1185c - 72 - r
0x1185d - 69 - i
0x1185e - 6e - n
0x1185f - 67 - g
0x11860 - 00 -
A jeśli zrobimy to samo na maszynie Intela jest to:
Adres - Kod - Litera

0x8048420 - 73 - s
0x8048421 - 69 - i
0x8048422 - 68 - h
0x8048423 - 74 - t
0x8048424 - 20 -
0x8048425 - 73 - s
0x8048426 - 69 - i
0x8048427 - 20 -
0x8048428 - 74 - t
0x8048429 - 73 - s
0x804842a - 20 -
0x804842b - 61 - a

0x804842c - 67 - g
0x804842d - 6e - n
0x804842e - 69 - i
0x804842f - 72 - r
Na pierwszy rzut oka architektura x86 można przeoczyć ,że jest to łańcuch jakiego szukamy. Jest to różnica w kolejności bajtów. W celu rozróżniania hostów w tej samej sieci , aby mogły komunikować się i wymieniać danymi, zgadzają się na wspólną kolejności bajtów. W nowoczesnych sieciach dana jest transmitowana w kolejności bajtów big endian tj,. Najbardziej znaczący bajt przychodzi pierwszy. W i80x86 porządek bajtu hosta to pierwszy Najmniej Znaczący Bajt (LSB), podczas gdy porządek bajtów w sieci, stosowany w Internecie to pierwszy Najbardziej Znaczący Bajt (MSB)

Czytanie assemblera

Śledzenie stosu i rejestrów

Tajemnicą zrozumienia kodu assemblera jest zawsze kawałek papieru i ołówek. Siadasz i najpierw rysujesz tabelkę dla 6 rejestrów A,B,C,D, SI i DI. Śledzimy również wysoką i niską część. Każda nowa linia tej tabeli powinna przedstawić zmodyfikowany rejestr, więc ostania wartość w każdej kolumnie rejestru jest bieżącą wartością tego rejestru. Następnie rysujemy długą kolumnę dla stosu, i zostaw miejsce po bokach na umieszczenie rejestrów BP i SP które przesuwają się w dół. Upewnij się ,że zapisałeś wszystkie wartości na stos tak jak one są tam umieszczone, wliczając w to ret i przechowywany BP.

Składnia AT&T

W składni AT&T wszystkie instrukcje są w postaci:
mnemonik źródło , przeznaczenie
Niezależne stałe numeryczne są poprzedzone znakiem $. Liczby szesnastkowe zawsze zaczynają się od 0x (w przeciwieństwie do kończącego h. Rejestry są poprzedzone znakiem %. tj %eax. Dereferencja i wskaźnik są przedstawiane w postaci disp(%baza, %indeks, skala), gdzie adres wynikowy to disp + %baza + %index* skala. disp i skala są stałymi (żadnego $) a %baza i %indeks s,a rejestrami. Każdy z tych czterech elementów może być pominięty, pozostawiając albo pustą przestrzeń a potem przecinek albo pozostaw to (argument, i wszystkie pozostałe argumenty. Na przykład, 4 (%eax) oznacza adres pamięci 4+%eax, gdzie (%eax,4) oznacza %eax * 4. Taka notacja tworzy łatwą tablicę indeksacji.

Zbiór instrukcji Intela

Tu jest kwestia zrozumienie co dany mnemonik asemblera robi. Większość mnemoników jest oczywista, ale znajdziesz pełny opis wszystkich instrukcji Intela w dokumentacji Intelowskiej. Zapamiętaj ,że w składni Intela, operandy są w odwrotnej kolejności niż w składni AT&T.

Poznaj swój kompilator

Aby nauczyć się skutecznie odczytywać assembler, rzeczywiście musisz wiedzieć jakiego typu kod twój kompilator generuje w pewnych sytuacjach. Jeśli nauczysz się rozpoznawać co to pętla, instrukcja if -else, możesz nauczyć się uzyskiwać ogólne czucie kodu dużo szybciej. Są również sztuczki, jakie wykonuje GCC, które mogą wydawać się nieintuicyjne na pierwszy rzut oka dla neofity reverse engineeringu, nawet jeśli już ma pojęcie jak działać w asemblerze.

Podstawowe struktury sterujące

W asemblerze tylko mechanizmy kontroli przepływu są rozgałęziane i wywoływane. Więc każda struktura sterująca jest oparta na połączeniu goto i rozgałęzień warunkowych. Spójrzmy na kilka określonych przykładów

Wywołania funkcji

Mówiłem wcześniej ,że wywołanie funkcji używa stosu do przekazania argumentów. Ale gdzie pozostawia wartość powrotną? I co ze zmiennymi lokalnymi? Zmienne lokalne również są na stosie, poniżej wskaźnika bazowego zamiast powyżej. Ale jeśli sądzisz ,że wartość powrotu została zdjęta ze stosu, to błąd! GCC umieszcza wartość powrotu określonej funkcji w rejestrze eax na końcu tej funkcji.

Instrukcja if

Instrukcja if jest przedstawiana w asemblerze jako test po której występuje jmp. Warto zauważyć ,że czasami ciało instrukcji if jest tym gdzie skacze, w przeciwieństwie do skoków przez kod C jaki możesz określić. Oznacza to ,że warunek dla skoków często będzie zanegowanym warunkiem twojej instrukcji if.

Instrukcja if...else

Widzimy ,że instrukcja if jest zazwyczaj robiona przez wykonanie prostego skoku przez ciało instrukcji. Instrukcje if...else działają w ten sam sposób, z wyjątkiem bezwarunkowego skoku na końcu ciała instrukcji if, która odwraca wykonanie przepływu na końcu ciała else.

Instrukcje If...else...if

Dodanie nowego if w klauzuli else działa w ten sam sposób jaki if wewnątrz klauzuli else. Po prostu skaczemy do innej etykiety jeśli wyliczają fałsz, a jeśli instrukcja if oblicza prawdę, na jej dole po prostu skaczemy do else if i pozostałych klauzuli else

Złożone instrukcje if

Oczywiście instrukcje if mogą być dużo bardziej złożone niż powyższe przykłady. Mogą zawierać boolowskei skróty, wywołania funkcji , zagnieżdżone if'y itp

Pętla while

Pomyśl przez sekundę o pętli while. Pomyśl o tym jak działa. W zasadzie, możesz napisać pętlę loop za pomocą instrukcji if i goto wewnątrz ciała if na górze pętli. Ponieważ jedynymi mechanizmami rozgałęziania jakie mamy w asemblerze są skoki i wywołania, pętle while są tylko instrukcjami if z jmp z powrotem na górę z dołu.

Pętla for

Przepiszmy powyższą pętle aby zobaczyć czy rzeczywiście te pętle są równoważne

Pętla do...while

Pętle do while są trochę inne niż pętle for i while, w tym ,że pozwalają na wykonanie ciała pętli przynajmniej raz. Porównanie ich instrukcji ma miejsce na dole pętli.

Tablice

Tablice na stosie

Tablice na stosie są tylko regionami pamięci, do których mamy dostęp z wariantami na disp (%baza, %indeks, skala) wspomnianymi wcześniej. Tablice jednowymiarowe są najłatwiejsze, ponieważ są proste, fragment pamięci jest liczbą elementów razy rozmiar każdego elementu. Tablice dwuwymiarowe są w rzeczywistości abstrakcyjne, która pracuje z pamięcią łatwiej niż w C. Tablica 2D na stosie jest tylko jedną długą tablicą 1D, którą kompilator C dzieli dla nas aby uczynić ją zarządzalną .Deklaracja tablicy: type array[dim2][dim1]; w rzeczywistości tablica 1D o długości dim2*dim1. Kompilator C obsługuje indeksowanie tablicy jak następuje: array[i][j] jest to komórka pamięci array +i*dim1*type + j* type. Dzielimy naszą tablicę 1D na sekcję dim2, każda długości dim1*type. Najlepszym sposobem myślenia o tablicach wielowymiarowych jest myślenie o nich jako zbiorze tablic klejnego niższego wymiaru. Więc myślenie o tym jak tablica 3D może być wbita w tablicę 1D jest myślenie o tym jak zbiór tablic 2D wbitych w tablicę 1D: jedna po drugiej. Dotychczas deklarowano tablicę jako type array[dim3][dim2][dim1];, array[i][j][k] oznacza array + i*dim2*dim1*type _j*dim1*type + k * type. Więc oznacza to po prostu mnożenie asemblerowe zmiennych indeksowych, co powinno pomóc określić n-1 wymiarów n wymiarowej tablicy. Pozostały wymiar może być określony z całkowitego rozmiaru, lub granicy jakiejś zainicjalizowanej pętli.

Struktury

Użycie struktur

Struktury (structs) to wygodny sposób na zarządzanie powiązanymi zmiennymi bez potrzeby pisania klasy zamykającej je wszystkie. Struktura jest zazwyczaj klasą bez żądnej funkcji składowej. Struktury są używane BARDZO często w C aby uniknąć przekazywania kilku zmiennych tam i z powrotem między funkcjami.. Zamiast tego przekazują wszystkie zmienne, wspólną praktyką jest zamknięcie wszystkich ich w strukturze przekazywać tylko położenie

struktury w pamięci do funkcji która potrzebuje dostępu do tych zmiennych. Struktura w C jest deklarowana tak:

```
struct a
{
int pierwsza;
float druga;
char *trzecia;
};
```

Nie zapomnij o ; po ostatnim nawiasie. Struktury mogą przechowywać typ zmiennej, które normalnie byłyby zadeklarowane gdzieśkolwiek w programie. Dostęp do zmiennych w strukturze odbywa się przez operator kropki (.). Na przykład, aby przypisać 5 do pierwszej zmiennej w strukturze a , zrób tak

```
a.first = 5;
```

Tablice struktur

Tablice struktur są tworzone tak jak tworzy się tablice innych zmiennych. Poniższa deklaracja tworzy tablicę struktur o rozmiarze 10 :

```
struct a structarray[1];
```

Zauważ zastosowanie słowa kluczowego struct, po którym występuje nazwa deklarowanej struktury a po niej nazwa tablicy. Powyższy kod deklaruje statyczną tablicę struktury. Oznacza to ,że przestrzeń będzie zaalokowana dla tej tablicy podczas czasu ładowania. Tablice struktur mogą również być deklarowane jako wskaźniki aby przestrzeń dla pojedynczych elementów może być zaalokowana w czasie wykonywania jeśli to konieczne. GCC obsługuje struktury nieco dziwnie. Kiedy masz funkcję , która zwraca strukturę, to co robi gcc w rzeczywistości odkłada adres struktury na stos przed wywołaniem tej funkcji (jak gdyby pierwszy argument tej funkcji był wskaźnik do tej struktury, która zawierała będzie wartość i powrotu) Potem, wewnątrz funkcji, kod jest generowany do modyfikacji struktury poprzez ten adres. Na końcu funkcji, wartość #eax zawiera wskaźnik do struktury, który został przekazany na stos. Więc zamiast normalnego przechowywania przez %eax wartości powrotu, %eax przechowuje wskaźnik do wartości powrotu, a wartość powrotu jest modyfikowana bezpośrednio wewnątrz tej funkcji.

V. DEBUGGING

Debuggowanie na poziomie użytkownika

gdb

gdb jest debuggerem GNU. Jest bardzo przerażający dla większości ludzi, ale rzeczywiście nie ma powodu aby tak było. Jest to dobrze zaprojektowany debugger dla wiersza poleceń. Jest też debugger z GUI zwany DDD ,ale dla naszych celów wybierzemy wiersz poleceń. gdb ma sympatyczny wbudowany system pomocy zorganizowany tematycznie. Wpisując help pokażą ci się kategorie. Główne polecenia jakimi będziemy się interesować to run ,break, cont, stepi, finish, dissassemble, bt, info [register/frame] i x. Każde polecenie w gdb może być zakończone liczbą N, która oznacza powtarzanie N razy. Na przykład stepi 1000, będzie powtarzał instrukcje asemblera 1000 razy.

Użycie windbg

WinDbg jest częścią standardowego Debugging Tools dla Windows, które można ściągnąć za darmo. Microsft oferuje kilka różnych debuggerów, które używaj wspólnych poleceń dla większości operacji i oczywiście są przypadki kiedy się różnią .Ponieważ WinDbg jest programem GUI, wszystkie działania są wykonywane przy użyciu dostarczonych komponentów wizualnych. Jest również wiersz poleceń osadzony w debuggerze, który pozwala na wpisywanie poleceń podobnie jak użycie debuggera konsolowego takiego jak ntsd

Punkty przerwań

Punkty przerwań mogą być ustawione, odznaczane lub listowane z GUI przez użycie Edit -> Breakpoints lub skrótu klawiaturowego Alt + F9. Z wiersza poleceń można ustawić punkt

przerwania używając polecenia bp, wylistować je z bl a usunąć poleceniem bc. Można ustawić punkt przerwania zarówno na nazwie funkcji lub komórce pamięci.

Debugowanie na poziomie jądra

Debuggowanie na poziomie jądra jest użyteczne jeśli chcesz spróbować odkryć jak działa określony sterownik, lub jeśli chcesz więcej informacji o określonym punkcie wejścia / API jądra. Niestety, obsługa debuggowania jądra jest dużo lepsze pod Windows niż pod Linuksem.

VI. FORMATY WYKONYWALNE

Praca z programem formatu ELF

W tym momencie wiemy już jak pisać programy niskopoziomowe, a zatem tworzyć plik wykonywalny bardzo ściśle dopasowany. Pytanie brzmi, jak kod naszego programu jest przechowywany na dysku? Cóż kiedy program się uruchamia, zaczynamy od funkcji _start i przechodzimy do _ libc _start _main, i ewentualnie do main, która jest naszym kodem. System operacyjny zbiera jakoś razem cały kod z różnych miejsc i ładuje go do pamięci a potem go uruchamia. Skąd wie jaki kod jest gdzie? Odpowiedzią jest binarna specyfikacja ELF na Linuksie i UNIX'ie. ELF określa standardowy format dla odwzorowania kodu na dysku dla zakończenia obrazu wykonywalnego w pamięci który składa się z kodu, stosu, sterty (dla malloc) i wszystkich bibliotek jakie łączysz. Więc omówimy informacje potrzebne do naszych celów .

Rozkład ELF

Są trzy obszary nagłówków w pliku ELF: Główny nagłówek pliku ELF, nagłówki programu i nagłówki sekcji. Kod programu leży między nagłówkami programu a nagłówkami sekcji. ELF jest elastyczny. Wiele z tej sekcji może być rozszerzane , usuwane itd.

Główny nagłówek pliku ELF

Główny nagłówek elf w zasadzie mówi nam gdzie wszystko jest umieszczone w tym pliku. Następuje na początku pliku wykonywalnego, i może być odczytany bezpośrednio z pierwszych bajtów e_ ehsize (domyślnie: 52) pliku w tej strukturze.

```
/* Nagłówek pliku ELF */
typedef struct
{
unsigned char e_ident[EI_NIDENT]; /* Magiczna liczba i inne info */
Elf32_Half e_type; /* Typ pliku obiektowego */
Elf32_Half e_machine; /* Architektura */
Elf32_Word e_version; /* Wersja pliku obiektowego */
Elf32_Addr e_entry; /* Punkt wejścia adresu wirtualnego */
Elf32_Off e_phoff; /* Offset tablicy pliku nagłówka programu */
Elf32_Off e_shoff; /* Offset tablicy pliku nagłówka sekcji */
Elf32_Word e_flags; /*Flagi procesora */
Elf32_Half e_ehsize; /* rozmair nagłówka ELF w bajtach*/
Elf32_Half e_phentsize; /* Rozmiar tablicy wejścia nagłówka programu*/
Elf32_Half e_phnum; /* Licznik tablicy wejścia nagłówka programu*/
Elf32_Half e_shentsize; /* Rozmiar tablicy wejścia nagłówka sekcji*/
Elf32_Half e_shnum; /* Licznik tablicy wejścia nagłówka sekcji*/
Elf32_Half e_shstrndx; /*Indeks tablicy łańcucha nagłówka sekcji*/
} Elf32_Ehdr;
```

Pola jakie nasz interesują to e_entry, e_phoff, e_shoff i podany rozmiar e_entry określa położenie _start, e _ phoff pokazuje gdzie leży tablica nagłówka programu w relacji do startu pliku wykonywalnego a e_shoff pokazuje to samo dla nagłówka sekcji.

Nagłówki programu

Kolejną częścią programu są nagłówki programu ELF. Opisują sekcję programu który zawiera kod programu wykonywalnego odwzorowane do przestrzeni adresowej programu podczas

ładowania.
/* Nagłówek segmentu programu. */
typedef struct
{
Elf32_Word p_type; /* Typ segmentu */
Elf32_Off p_offset; /* Offset Segmentu pliku */
Elf32_Addr p_vaddr; /* Adres wirtualny Segmentu */
Elf32_Addr p_paddr; /* Adres fizyczny segmentu */
Elf32_Word p_filesz; /* Rozmiar segmentu w pliku */
Elf32_Word p_memsz; /* Rozmiar segmentU w pamięci */
Elf32_Word p_flags; /* Flagi segmentu */
Elf32_Word p_align; /* Wyrównanie segmentu */
} Elf32_Phdr;
Interesujące pola w tej strukturze to p_offset, p_filesz i p_memsz.

Ciało ELF

Aktualne położenie i rozmiary części ciała są opisane przez powyższy nagłówek programu, i zawiera instrukcje wykonywalne z naszego pliku asemblowanego, jak również stały łańcuch i deklarację zmiennej globalnej.

Nagłówki sekcji ELF

Nagłówki sekcji ELF opisują różne nazwane sekcje w pliku wykonywalnym. Każda sekcja ma wejście w tablicy nagłówków sekcji, znajdującą się na dole pliku wykonywalnego i mają następujący format:
/* Nagłówek sekcji */
typedef struct
{
Elf32_Word sh_name; /* Nazwa sekcji*/
Elf32_Word sh_type; /* Typ sekcji */
Elf32_Word sh_flags; /* Flagi sekcji */
Elf32_Addr sh_addr; /* Adres wirtualny sekcji przy wykonaniu*/
Elf32_Off sh_offset; /* Offset sekcji pliku */
Elf32_Word sh_size; /* Rozmiar sekcji w bajtach */
Elf32_Word sh_link; /* Łącze do innej sekcji */
Elf32_Word sh_info; /* Dodatkowe informacje o sekcji*/
Elf32_Word sh_addralign; /* Wyrównanie sekcji */
Elf32_Word sh_entsize; /*Rozmiar wejścia jeśli sekcja przechowuje tablicę*/
} Elf32_Shdr;

Edytowanie ELF

Edytowanie ELF jest pożądane podczas reverse engineeringu, szczególnie kiedy chcemy wstawić ciało do kodu, lub jeśli chcemy odwrócić binaria z celowo uszkodzonych nagłówków ELF. Teraz możesz edytować te nagłówki przez użycie pliku nagłówkowego i powyższych struktur, ale szczęśliwie jest już edytor nazwany HT Editor, który pozwala na zbadanie i modyfikację wszystkich sekcji programu ELF, z nagłówka ELF do rzeczywistych instrukcji.

VII. MODYFIKACJA KODU

Teraz znamy narzędzia do analizy naszych programów i znajdowania funkcji nas interesujących nawet w programach bez kodu źródłowego. Możemy zrozumieć asembler, i pisać w asemblerze do naszych własnych celów. Wiemy jak program wygląda na dysku i jak wygląda w pamięci. Wiedza jest potęgą a my wiemy dużo. Powody modyfikacji kodu Modyfikacja kodu jest najbardziej użyteczna jeśli życzysz sobie zmian zachowania programów z zamkniętym kodem napisanych przez nieuświadomionych autorów. Jest poręczna również kiedy próbujesz zajmować się różnego rodzajami zabezpieczeniami przed kopiowaniem.

Biblioteka Hooking

LD_PRELOAD

Jest to zmienna środowiskowa, która pozwala nam dodawać bibliotekę do wykonywania określonego programu. Dowolna funkcja w tej bibliotece automatycznie przykrywa funkcje standardowej biblioteki.

Modyfikacje instrukcji

Ponieważ najmniejszą jednostką kodu jest instrukcja, najprostszą formą modyfikacji kodu jest modyfikacja instrukcji. W modyfikacji instrukcji, dokonujemy zmian jakiejś właściwości określonej instrukcji. Jak pamiętasz z sekcji o asemblerze każda instrukcja ma 2 części : mnemonik i argumenty. Więc nasz wybór jest ograniczony. Najlepszym sposobem modyfikacji instrukcji jest HT Editor. HTE ma tryb edytora szesnastkowego gdzie możesz edytować wartości szesnastkowe instrukcji i zobaczyć asemblację w czasie rzeczywistym.

Edycja argumentów

Edytowanie argumentów asemblowanych instrukcji jest łatwe. Po porostu patrzysz na wartość szesnastkową argumentu asemblowanej instrukcji i widzisz gdzie leży bajt heksadecymalny tej instrukcji. HTE pozwoli ci nadpisać te wartości swoimi własnymi (Bądź ostrożny przy porządkowaniu bajtu)