Tworzenie środowiska pracy

Jeśli wykorzystujesz przepełnienia i ciągi formatujące oraz inne problemy na poziomie kodu powłoki, potrzebujesz dobrego środowiska pracy. Przez otoczenie nie mam na myśli zaciemnionego pokoju z dużą ilością pizzy i dietetycznych napojów gazowanych. Odnoszę się do dobrego zestawu narzędzi do kodowania, narzędzi do śledzenia i materiałów referencyjnych, które pomogą Ci wykonać zadania przy minimalnym zamieszaniu. Ta Część da ci punkt wyjścia do ustalenia tego środowiska. Ogólnie rzecz biorąc, jeśli chcesz wykorzystać błąd, potrzebujesz co najmniej dwóch elementów: zestawu dokumentów referencyjnych i podręczników, które dostarczają potrzebnych informacji na temat systemu, który wykorzystujesz, oraz zestawu narzędzi do kodowania, dzięki którym możesz pisać exploit. Dodatkowo bardzo przydatny jest zestaw narzędzi, które można wykorzystać do śledzenia (dokładnej obserwacji testowanego systemu). Zaczniemy od krótkiego przeglądu najpopularniejszych przedmiotów w każdej z tych trzech kategorii. Ponieważ coś nowego pojawia się w świecie shellcode prawie codziennie, nie traktuj tego jako najnowocześniejszej, najnowocześniejszej dyskusji na temat tego, co tam jest; jest to raczej szybkie kompendium najlepszych referencji, narzędzi do kodowania i narzędzi do śledzenia dostępnych w momencie pisania. Ponadto nie faworyzujemy konkretnego systemu operacyjnego, więc nie wszystkie wymienione elementy będą dotyczyć systemu operacyjnego, na który kierujesz reklamy. Podaję odpowiedni system operacyjny, jeśli jest to ważne – jeśli nie ma na liście żadnego systemu operacyjnego, to albo jest to narzędzie, które działa praktycznie na wszystkim, albo jest to dokument odnoszący się do ogólnej klasy problemów.

Czego potrzebujesz do pisania kodu

Aby napisać swój kod, potrzebujesz narzędzi. Poniżej znajduje się krótkie omówienie niektórych z bardziej popularnych narzędzi wśród szelkoderów x86.

gcc : gcc (GNU Compiler Collection) to w rzeczywistości znacznie więcej niż kompilator C/C++; gcc zawiera również interfejsy dla Fortran, Java i Ada. Jest to prawie na pewno najlepszy dostępny darmowy (GPL) kompilator, a dzięki obsłudze asemblacji inline jest doskonałym wyborem dla programisty shellcode.

gdb : gdb (GNU Debugger) to darmowy (GPL) debugger, który dobrze integruje się z gcc i zapewnia oparte na wierszu poleceń symboliczne środowisko debugowania. Posiada również doskonałe wsparcie dla interaktywnego demontażu i dlatego jest dobrym wyborem do badania początkowych wektorów pod kątem błędu przepełnienia/formatu łańcucha.

NASM : NASM (Netwide Assembler) to darmowy assembler x86 obsługujący różne wyjściowe formaty plików binarnych, takie jak Linux i BSD a.out, ELF, COFF oraz 16- i 32-bitowe formaty obiektów i plików wykonywalnych Windows. NASM to niezwykle przydatne narzędzie, jeśli potrzebujesz dedykowanego asemblera. W swojej dokumentacji ma również doskonałe odniesienie do kodu operacji x86.

WinDbg : WinDbg to samodzielny debugger dla platformy Windows dostarczany przez firmę Microsoft. Posiada przyjazny interfejs GUI z wieloma doskonałymi funkcjami, w tym wyszukiwaniem w pamięci, możliwością debugowania procesów podrzędnych i rozbudowanymi funkcjami obsługi wyjątków. WinDbg jest przydatne, jeśli chcesz napisać exploita dla programu na platformie Windows, który uruchamia procesy potomne (takich jak Oracle lub Apache), ponieważ może automatycznie podążać za nimi i dołączać do nich.

OllyDbg : OllyDbg to „debugger analizujący” systemu Windows. OllyDbg zawiera niezwykle fajne funkcje, takie jak pełne przeszukiwanie pamięci (brak tego w WinDbg) i świetny deasembler.

Korzystanie z OllyDbg przypomina posiadanie większości najlepszych części WinDbg i IDA w jednym, darmowym narzędziu.

Visual C++ : Visual C++ to flagowy kompilator C/C++ firmy Microsoft. Ma doskonały interfejs użytkownika i wbudowane pełne funkcje debugowania. Visual C++ w pełni integruje się z zestawem dokumentacji Microsoft Developer Network (MSDN), co może być niezwykle przydatne, jeśli piszesz exploity dla systemu Windows — mając zintegrowane dobre odniesienie do interfejsu API Win32 do twojego IDE znacznie przyspiesza pracę. Podobnie jak gcc, Visual C++ obsługuje asembler inline, co upraszcza tworzenie exploitów. Podsumowując, jeśli masz dostęp do licencji na Visual C++/Developer Studio, to warto zajrzeć.

Python : Ostatnio wielu programistów exploitów pisało swoje exploity w Pythonie, języku dobrze znanym z szybkiego tworzenia aplikacji. . Po dodaniu MOSDEF, czystego asemblera Pythona i narzędzia do programowania shellcode, Python może być jednym z najskuteczniejszych narzędzi w Twoim arsenale.

Czego potrzebujesz do dochodzenia

Aby znaleźć błędy, będziesz potrzebować dobrego rozeznania, co dzieje się w wewnętrznych strukturach aplikacji lub programu, który atakujesz. Narzędzia wymienione w tej sekcji są przydatne w różnych sytuacjach, takich jak polowanie na błędy, opracowywanie exploita i próba sprawdzenia, co robi ktoś inny.

Przydatne niestandardowe skrypty/narzędzia

Oprócz narzędzi wymienionych tu, używamy różnych małych, niestandardowych narzędzi do różnych celów. Możesz napisać własne skrypty lub narzędzia do podobnych celów.

Wyszukiwarka offsetów

Na platformach Windows i Unix często będziesz musiał znaleźć adres danej instrukcji. Na przykład w przypadku przepełnienia stosu systemu Windows może się okazać, że rejestr ESP wskazuje na twój kod powłoki. Aby to wykorzystać, musisz znaleźć adres jakiegoś strumienia instrukcji, który przekierowuje wykonanie do twojego kodu. Najprostszym sposobem, aby to zrobić, jest znalezienie w pamięci jednej z następujących sekwencji bajtów, a następnie nadpisanie zapisanego adresu powrotu adresem jednej z następujących sekwencji:

jmp esp (0xff 0xe4)

call esp (0xff 0xd4)

push esp; ret (0x54 0xc3)

Powinieneś znaleźć te sekwencje w wielu miejscach w pamięci. Najlepiej byłoby szukać ich w bibliotekach DLL, które nie uległy zmianie w dodatkach Service Pack. Wyszukiwarka offsetów zazwyczaj działa poprzez dołączenie do zdalnego procesu i zawieszenie wszystkich jego wątków, a następnie szukanie w pamięci określonych sekwencji bajtów, raportowanie ich do pliku tekstowego. To prosta, ale przydatna rzecz. Alternatywnie projekt Metasploit ma bazę danych opcode online pod adresem http://www.metasploit.com/opcode_database.html.

Ogólne Fuzzery

Jeśli badasz dany produkt pod kątem luk w zabezpieczeniach, prawdopodobnie przyda Ci się napisanie fuzzera, który skupia się na określonej funkcji produktu — interfejsie internetowym lub

niestandardowym protokole sieciowym, a może nawet interfejsie RPC. Ponownie, ogólne fuzzery to przydatne rzeczy, które warto mieć pod ręką. Nawet bardzo proste fuzzery mogą wiele zdziałać.

Sztuczka debugowania

Odwrócone powłoki w systemie Windows mogą być dość frustrujące. Nie można łatwo przesyłać plików, a podstawowa obsługa skryptów jest ograniczona. Jednak w tym wilgotnym i przerażającym świecie jest promyk nadziei, który pojawia się w mało prawdopodobnej formie starego debuggera MS-DOS, debug.exe. Debug.exe znajdziesz w prawie każdym systemie Windows. Istnieje od czasu MS-DOS i nadal istnieje w najnowszej wersji systemu Windows XP. Chociaż debug.exe był pierwotnie przeznaczony do debugowania i tworzenia plików .com, można go również używać do tworzenia dowolnego pliku binarnego — z pewnymi ograniczeniami. Plik musi być mniejszy niż 64 KB, a nazwa pliku nie może kończyć się na .exe ani .com. Weźmy na przykład następujący plik binarny:

73 71 75 65 61 6D 69 73 68 20 6F 73 73 69 66 72 squeamish ossifr

61 67 65 0A DE DE DE DE DE DE DE CO DE Wiek@p@p@p@p@p@p@p@p

Możesz napisać plik skryptu, który wyprowadza wspomniany plik binarny w następujący sposób (nazwij go foo.scr):

n foo.scr

e 0000 73 71 75 65 61 6d 69 73 68 20 6f 73 73 69 66 72

e 0010 61 67 65 0d 0a de c0 de de c0 de de c0 de

rcx

1e

w 0

Q

Następnie uruchom debug.exe.

debug < foo.scr

debug.exe wygeneruje plik binarny. Chodzi o to, że plik skryptu musi zawierać tylko znaki alfanumeryczne, aby można było użyć polecenia echo w odwrotnej powłoce, aby go utworzyć. Gdy plik skryptu znajduje się na zdalnym hoście, uruchamiasz debug.exe w określony sposób i bingo, masz swój plik binarny. Możesz po prostu zmienić nazwę początkowego pliku, na przykład nc.foo, a następnie (po przesłaniu) zmienić nazwę na nc.exe. Jedyną rzeczą, którą musisz zautomatyzować, jest tworzenie pliku skryptu. Po raz kolejny można to łatwo zrobić w perlu, Pythonie lub C.debug.exe jest wyjątkowo przydatnym narzędziem, jeśli nalegasz na używanie odwróconych powłok w systemie Windows. Istnieją inne sposoby na przesyłanie plików binarnych w systemie Windows — na przykład możliwe jest utworzenie pliku .com składającego się wyłącznie ze znaków drukowalnych, które można wykorzystać do utworzenia dowolnego pliku binarnego.

Wszystkie platformy

Prawdopodobnie najpopularniejszym istniejącym narzędziem do zabezpieczania sieci, które może być używane na wszystkich platformach, jest NetCat. Jego pierwotny autor, Hobbit, opisał Net-Cat jako swój „szwajcarski scyzoryk TCP/IP”. NetCat umożliwia wysyłanie i odbieranie dowolnych danych na dowolnych portach TCP i UDP, a także nasłuchiwanie (na przykład) odwróconych powłok. NetCat jest

standardowo dostarczany z kilkoma dystrybucjami Linuksa i ma port Windows. Jest nawet wersja GNU .Oryginalne wersje dla systemów Unix i Windows — autorstwa Hobbita i Chrisa Wysopala (Weld Pond)

Unix

Ogólnie rzecz biorąc, łatwiej jest zobaczyć, co się dzieje w systemie Unix niż w systemie Windows; dlatego łowcom błędów jest to nieco łatwiejsze. ltrace i strace ltrace i strace to programy, które umożliwiają przeglądanie wywołań biblioteki dynamicznej i wywołań systemowych, które tworzy program, a także przeglądanie sygnałów odbieranych przez program. ltrace jest wyjątkowo przydatne, jeśli próbujesz dowiedzieć się, jak konkretna część pewnego mechanizmu obsługi ciągów działa w procesie docelowym. strace jest również bardzo przydatne, jeśli próbujesz uniknąć IDS opartego na hoście i musisz ustalić, jaki wzór wywołań systemowych tworzy program.

truss

truss zapewnia mniej więcej taką samą funkcjonalność, jak połączenie ltrace i strace w systemie Solaris.

fstat (BSD)

fstat to oparte na BSD narzędzie do identyfikowania otwartych plików (w tym gniazd). Jest to bardzo przydatne do szybkiego sprawdzania, które procesy robią co w złożonym środowisku.

tcpdump

Ponieważ najlepsze błędy to błędy zdalne, sniffer pakietów jest niezbędny. tcpdump może być przydatny do uzyskania szybkiego przeglądu tego, co robi dany demon; dla bardziej szczegółowej analizy jednak Wireshark (omawiany dalej) jest prawdopodobnie lepszy.

Wireshark (Ethereal)

Wireshark to oparty na GUI darmowy sniffer i analizator pakietów sieciowych. Ma ogromną liczbę parserów pakietów, więc jest całkiem dobrym pierwszym wyborem, jeśli próbujesz zrozumieć nietypowy protokół sieciowy lub piszesz fuzzer protokołu.

Windows

Życie łowcy błędów na platformie Windows jest nieco trudniejsze. Poniższe narzędzia są wyjątkowo przydatne - wszystkie można pobrać ze znakomitej witryny Sysinternals autorstwa Marka Russinovicha i Bryce′a Cogswella, teraz przeniesionej do firmy Microsoft.

* RegMon - monitoruje dostęp do rejestru systemu Windows za pomocą filtra, dzięki czemu można skoncentrować się na testowanych procesach.

* FileMon - monitoruje aktywność plików, ponownie z przydatną funkcją filtrowania.

* HandleEx - wyświetla biblioteki DLL załadowane przez proces, a także wszystkie otwarte uchwyty; na przykład nazwane potoki, sekcje pamięci współdzielonej i pliki.

* TCPView — kojarzy punkty końcowe TCP i UDP z procesem, który jest ich właścicielem.

* Eksplorator procesów — umożliwia badanie w czasie rzeczywistym procesów, uchwytów, bibliotek DLL i nie tylko Witryna Sysinternals zapewnia wiele doskonałych narzędzi, ale te pięć programów stanowi dobry zestaw narzędzi na początek.

Deasembler IDA Pro

Deasembler IDA pro jest najlepszym narzędziem do demontażu na rynku dla inżyniera wstecznego Windows. Posiada doskonały, skryptowalny interfejs użytkownika z łatwymi funkcjami odsyłania i wyszukiwania. IDA Pro jest szczególnie przydatne, gdy musisz dokładnie ustalić, co robi dany podatny kod i gdy masz problemy z takimi zadaniami, jak kontynuacja wykonywania lub kradzież gniazd. IDA można znaleźć na stronie www.datarescue.com/.

Optymalizacja rozwoju kodu Shell

Pierwszy exploit, który napiszesz, będzie najtrudniejszy i najbardziej żmudny. W miarę gromadzenia kolejnych exploitów i większego doświadczenia nauczysz się optymalizować różne zadania, aby skrócić czas między znalezieniem błędu a uzyskaniem ładnie zapakowanego exploita. Ta sekcja jest krótką próbą wydestylowania naszych technik w krótki, czytelny przewodnik po optymalizacji procesu rozwoju. Oczywiście najlepszym sposobem na przyspieszenie rozwoju szelkodu jest nie pisanie szelkodu -użyj zamiast tego syscall proxy lub mechanizmu proglet. Jednak w większości przypadków prosty, statyczny exploit jest najłatwiejszy do zrobienia, więc zacznijmy rozmawiać o tym, jak to zoptymalizować i poprawić jego jakość.

Zaplanuj wykorzystanie

Zanim zaczniesz na ślepo rzucić się do pisania exploita, dobrze jest mieć solidny plan kroków, które podejmiesz, aby wykorzystać błąd. W przypadku przepełnienia stosu waniliowego na platformie Windows plan może wyglądać następująco (w zależności od tego, jak osobiście napisałbyś tego rodzaju exploit):

1. Określ przesunięcie bajtów nadpisujących zapisany adres powrotu.

2. Określ lokalizację ładunku względem rejestrów. (Czy ESP wskazuje na nasz bufor? Jakieś inne rejestry?)

3. Znajdź wiarygodne przesunięcie jmp/call <rejestr> dla (a) wersji produktu lub (b) różnych docelowych wersji systemu Windows i dodatków Service Pack.

4. Utwórz mały testowy kod powłoki nops, aby ustalić, czy ma miejsce korupcja.

5. Jeśli występuje uszkodzenie, wstaw jmps do ładunku, aby uniknąć uszkodzonych obszarów.

Jeśli nie ma korupcji, zastąp rzeczywisty szelkod.

Napisz Shellcode w Inline Assemblerze

Ta sztuczka może zaoszczędzić ci dużo czasu. Większość opublikowanych exploitów zawiera niezrozumiałe strumienie bajtów szesnastkowych zakodowanych w stałych łańcucha C. Nie pomaga to, jeśli musisz wstawić jmp, aby uniknąć uszkodzenia części stosu lub jeśli chcesz dokonać szybkiej modyfikacji kodu powłoki. Zamiast stałych C wypróbuj coś takiego (ten kod dotyczy Visual C++, ale podobna technika działa w przypadku gcc):

```
char *sploit()

{

asm

{

; this code returns the address of the start of the code
```

```
jmp get_sploit

get_sploit_fn:

pop eax

jmp got_sploit

get_sploit:

call get_sploit_fn ; get the current address into eax

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Exploit

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; start of exploit

jmp get_eip

get_eip_fn:

pop edx

jmp got_eip

get_eip:

call get_eip_fn ; get the current address into edx

call_get_proc_address:

mov ebx, 0x01475533 ; handle for loadlibrary

sub ebx, 0x01010101

mov ecx, dword ptr [ebx]
```

I tak dalej. Pisanie kodu w ten sposób ma kilka zalet:

* Możesz łatwo komentować kod asemblera, co jest pomocne, gdy musisz zmodyfikować swój kod powłoki sześć miesięcy później.

* Możesz debugować szelkod i testować go z komentarzami i łatwym wykrywaniem przerwań, bez faktycznego odpalania exploita. Breakpointing jest przydatny, jeśli twój exploit nie tylko tworzy powłokę.

* Możesz łatwo wycinać i wklejać sekcje kodu powłoki z innych exploitów.

* Nie musisz wykonywać skomplikowanych ćwiczeń polegających na wycinaniu i wklejaniu za każdym razem, gdy chcesz zmienić kod — po prostu zmień asembler i uruchom go.

Oczywiście pisanie exploita w ten sposób zmienia nieco uprząż, więc potrzebujesz metody na określenie długości exploita. Jedną z metod jest unikanie instrukcji, których wynikiem są bajty null, a następnie wklejanie instrukcji na końcu kodu powłoki.

```
add bajt ptr [eax],al
```

Pamiętaj, że poprzednie składa się z dwóch bajtów null. Uprząż może wtedy po prostu zrobić strlen, aby znaleźć długość exploita.

Utrzymuj bibliotekę Shellcode

Najszybszym sposobem na napisanie kodu jest wycinanie i wklejanie kodu, który już działa. Jeśli piszesz exploit, nie ma znaczenia, czy kod, który wycinasz, należy do Ciebie, czy kogoś innego, o ile dokładnie rozumiesz, co robi. Jeśli nie rozumiesz, co robi fragment szelkodu, na dłuższą metę prawdopodobnie szybciej jest napisać coś, co wykona to zadanie samodzielnie, ponieważ wtedy będziesz mógł go łatwiej modyfikować. Gdy będziesz mieć kilka działających exploitów, będziesz miał tendencję do używania tych samych ogólnych ładunków, ale zawsze warto mieć w pobliżu inne, bardziej złożone kody, aby ułatwić sobie korzystanie z nich. Wystarczy umieścić fragmenty kodu w łatwej do przeszukiwania formie, takiej jak hierarchia katalogów zawierających pliki tekstowe. Szybki grep i masz kod, którego potrzebujesz.

Niech to będzie ładnie kontynuowane

Kontynuacja wykonywania to wyjątkowo złożony temat, ale jest kluczem do pisania wysokiej jakości exploitów. Oto krótka lista podejść do problemu wraz z innymi przydatnymi informacjami:

* Jeśli zakończysz proces docelowy, czy zostanie on ponownie uruchomiony? Jeśli tak, wywołaj exit() lub ExitProcess() lub TerminateProcess() w systemie Windows.

* Jeśli zakończysz wątek docelowy, czy zostanie on ponownie uruchomiony? Jeśli tak, wywołaj ExitThread(), TerminateThread() lub odpowiednik. Ta metoda działa bardzo dobrze, jeśli używasz DBMS, ponieważ mają tendencję do używania pul wątków roboczych. (Oracle i SQL Server to robią.)

* Jeśli masz przepełnienie sterty, czy możesz ją naprawić? To trochę trudne, ale ta książka zawiera kilka dobrych wskazówek.

Jeśli chodzi o przywracanie przepływu kontroli, masz kilka alternatyw:

* Uruchom procedurę obsługi wyjątków. Najpierw sprawdź, czy nie ma obsługi wyjątków, zgodnie z ogólną zasadą, że najłatwiejszym do napisania kodem jest kod, którego nie piszesz. Jeśli proces docelowy ma już w pełni funkcjonalny program obsługi wyjątków, który ładnie czyści i restartuje wszystko, dlaczego po prostu go nie wywołać lub nie wywołać, powodując wyjątek?

* Napraw stos i wróć do rodzica. Ta technika jest trudna, ponieważ prawdopodobnie na stosie znajdują się informacje, których nie można łatwo uzyskać, przeszukując pamięć. Jednak ta metoda jest możliwa w niektórych przypadkach. Zaletą jest to, że możesz zapewnić, że nie wystąpi wyciek zasobów. Zasadniczo znajdujesz części stosu, które zostały nadpisane po przejęciu kontroli, przywracasz je do wartości sprzed przejęcia kontroli i uruchamiasz ret.

* Powrót do przodka. Zwykle możesz zastosować tę metodę, dodając stałą do stosu i wywołując ret. Jeśli zbadasz stos wywołań w punkcie, w którym uzyskujesz kontrolę, prawdopodobnie znajdziesz w drzewie wywołań jakiś punkt, do którego możesz wrócić bez problemu. Działa to dobrze na przykład w przypadku błędu SQL-UDP (który był używany przez robaka SQL Slammer). Prawdopodobnie jednak wycieknie z niektórych zasobów.

* Zadzwoń do przodka. W skrócie, możesz być w stanie po prostu wywołać procedurę wysoko w drzewie wywołań, na przykład procedurę głównego wątku. W niektórych aplikacjach działa to ładnie. Minusem jest to, że prawdopodobnie wycieknie wiele zasobów (gniazd, pamięci, uchwytów plików), które mogą później spowodować niestabilność programu.

Spraw, aby exploit był stabilny

Dobrym pomysłem jest zadanie sobie serii pytań po uruchomieniu exploita, aby móc określić, czy musisz dalej pracować, aby uczynić go bardziej stabilnym. Chociaż niektórym czytelnikom może wystarczyć wskazanie pojedynczego działającego przykładu exploita, jeśli faktycznie zamierzasz używać go w środowisku produkcyjnym, powinieneś dążyć do przemysłowych exploitów, które będą działać wszędzie i nie zmieni hosta docelowego w żadnym niepożądanym sposobie. Jest to ogólnie dobry pomysł, ale pomaga również skrócić ogólny czas rozwoju; jeśli wykonasz dobrą robotę za pierwszym razem, nie będziesz musiał poprawiać swojego exploita za każdym razem, gdy pojawi się problem. Oto krótka lista; możesz dodać więcej własnych pytań:

* Czy możesz wykorzystać swój exploit przeciwko hostowi więcej niż raz?

* Jeśli oskryptujesz swój exploit i będziesz go wielokrotnie uruchamiał na jednym hoście, czy w pewnym momencie się nie powiedzie? Czemu?

* Czy możesz jednocześnie uruchomić wiele kopii swojego exploita na hoście?

* Jeśli masz exploita systemu Windows, czy działa on we wszystkich dodatkach Service Pack docelowego systemu operacyjnego?

* Czy działa w innych systemach operacyjnych Windows? NT/2000/XP/2003?

* Jeśli masz exploit Linuksa, czy działa on w wielu dystrybucjach? Bez konieczności określania offsetów/wersji?

* Jeśli chcesz, aby użytkownicy wprowadzili zestaw przesunięć, aby Twój exploit działał, rozważ zakodowanie na stałe zestawu wspólnych przesunięć platformy w swoim exploicie i umożliwienie użytkownikowi wyboru na podstawie przyjaznej nazwy. Co więcej, użyj techniki, która sprawia, że exploit jest bardziej niezależny od platformy, na przykład wyprowadzanie adresów LoadLibrary i GetProcAddress z nagłówka PE w systemie Windows lub nie poleganie na zachowaniach specyficznych dla dystrybucji w systemie Linux.

* Co się stanie, jeśli host docelowy ma dobrze skonfigurowany skrypt zapory? Czy Twój exploit zawiesza docelowego demona, jeśli IPTables lub (w systemie Windows) zestaw reguł filtrowania IPSec blokuje połączenie?

* Jakie dzienniki pozostawia i jak można je wyczyścić? Spraw, aby ukraść połączenie Jeśli wykorzystujesz błąd zdalny (a jeśli nie, dlaczego nie?), najlepiej jest ponownie użyć połączenia, na którym przyszedłeś, tego dla swojej powłoki, strumienia danych syscall proxy i tak dalej. Oto kilka wskazówek, jak to zrobić:

* Przerwij wspólne wywołania gniazd — akceptuj, odbieraj, odbieraj z, wyślij, wyślij do — i sprawdź, gdzie jest przechowywany uchwyt gniazda. W swoim szelkodzie przeanalizuj uchwyt i użyj go ponownie. Może to obejmować użycie określonego przesunięcia stosu lub ramki lub ewentualnie brutalnego wymuszania za pomocą getpeername, aby znaleźć gniazdo, z którym rozmawiasz.

* W systemie Windows warto również zastosować punkt przerwania ReadFile i WriteFile, ponieważ są one czasami używane w uchwytach gniazda.

* Jeśli nie masz wyłącznego dostępu do gniazdka, nie poddawaj się. Dowiedz się, jak dostęp do gniazda jest serializowany i zrób to samo. Na przykład w systemie Windows docelowy proces prawdopodobnie będzie używał zdarzenia, semafora, muteksu lub sekcji krytycznej. W pierwszych trzech przypadkach wątki, o których mowa, prawdopodobnie będą wywoływać WaitForSingleObject(Ex) lub WaitForMultipleObjects(Ex), aw drugim przypadku muszą wywoływać EnterCriticalSection. We

wszystkich tych przypadkach, po ustaleniu uchwytu (lub sekcji krytycznej), na którą wszyscy czekają, możesz sam poczekać na dostęp i dobrze grać z innymi wątkami.