Inne platformy - Windows, Solaris, OS/X i Cisco

Teraz, gdy ukończyłeś sekcję wprowadzającą dotyczącą rozwoju luk w zabezpieczeniach platformy Linux/IA32, zbadamy trudniejsze i bardziej podchwytliwe systemy operacyjne oraz koncepcje eksploatacji. Przenosimy się do świata Windows, gdzie szczegółowo opisujemy kilka interesujących koncepcji eksploatacji z punktu widzenia hakera Windows. Pierwsza sekcja, pomoże ci zrozumieć, czym Windows różni się od zawartości Linux/IA32 w Części I. W kolejnej sekcji przechodzimy od razu do shellcodu Windows, a następnie zagłębiamy się w bardziej zaawansowaną zawartość Windows . Na koniec podsumujemy zawartość Windows sekcją o pokonywaniu filtrów dla Windows. Koncepcje obchodzenia różnych filtrów można zastosować w każdym scenariuszu wrogiego wstrzykiwania kodu. Pozostałe sekcje pokazują, jak wykrywać i wykorzystywać luki w zabezpieczeniach systemów operacyjnych Solaris i OS X oraz platformy Cisco. Ponieważ Solaris działa na zupełnie innej architekturze niż opisana do tej pory zawartość Linuksa i Windowsa, może na pierwszy rzut oka wydawać się obcy. W dwóch sekcjach Solaris będziesz hackował Solaris na SPARC jak mistrz, wprowadzając platformę Solaris i zagłębiając się w bardziej zaawansowane koncepcje, takie jak nadużywanie Tabeli Powiązań Procedury i użycie natywnego szyfrowania Blowfish w shellcodzie. Kolejna sekcja przedstawia OS X i omawia specyfikę pisania exploitów na platformach Intel i PowerPC. Kolejna sekcja omawia różne platformy i techniki Cisco, które mogą pomóc w znajdowaniu i wykorzystywaniu błędów, i omawia różne mechanizmy ochrony przed exploitami, które zostały niedawno (i w niektórych przypadkach nie tak niedawno) wprowadzone do większości popularnych systemów operacyjnych i kompilatorów. Po ukończeniu powinieneś opanować większość technik potrzebnych do zrozumienia i pisania exploitów w praktycznie każdym systemie operacyjnym, a także dogłębne zrozumienie różnych przeszkód, jakie napotyka system operacyjny i kompilator. sprzedawcy przeszkadzają.

Dziki świat Windows

Osiągnęliśmy punkt, w którym wszystkie systemy operacyjne będą definiowane przez różnice w stosunku do Linuksa. Ta sekcja da doświadczonym hakerom Windows świeże spojrzenie na problemy Microsoftu, a jednocześnie pozwoli hakerom zorientowanym na Uniksa na dobre zrozumienie elementów wewnętrznych Windows. Pod koniec powinieneś być w stanie napisać podstawowy exploit dla systemu Windows i uniknąć niektórych typowych pułapek, które staną ci na drodze, gdy spróbujesz bardziej złożonych exploitów. Dowiesz się również, jak korzystać z podstawowych narzędzi do debugowania systemu Windows. Po drodze zdobędziesz wiedzę na temat zabezpieczeń i modelu programowania systemu Windows oraz podstawową wiedzę na temat Distributed Component Object Model (DCOM) i Portable Executable-Common File Format (PE-COFF). Krótko mówiąc, zawiera wszystko, co haker na poziomie eksperta z wieloletnim doświadczeniem w świecie rzeczywistym chciałby wiedzieć, gdy po raz pierwszy uczy się atakować platformy Windows.

Czym Windows różni się od Linuksa?

Zespół Windows NT podjął na początku kilka decyzji projektowych, które miały głęboki wpływ na każdą powstałą architekturę. Projekt NT był w pełnym rozkwicie w 1989 roku, a jego pierwsze wydanie miało miejsce w 1991 roku jako Windows NT 3.1. Większość elementów wewnętrznych pierwotnie była inspirowana VMS, chociaż istniało kilka głównych różnic między VMS a NT, w szczególności włączenie wątków jądra we wczesnych wersjach jądra NT. Ta sekcja omawia niektóre główne cechy NT, które mogą nie być rozpoznawalne dla kogoś przyzwyczajonego do wewnętrznych elementów Linuksa lub Uniksa.

Win32 API i PE-COFF

OllyDbg, w pełni funkcjonalny debugger analizujący na poziomie asemblera, działający w systemie Windows , jest potężnym narzędziem do analizy binarnej. Zawartość tej części najlepiej zrozumiesz podczas pracy z debugerem analizy binarnej, takim jak OllyDbg. Aby zastosować to, czego się tutaj uczysz, będziesz potrzebować narzędzia z jego funkcjami. OllyDbg jest rozpowszechniany na licencji shareware i można go znaleźć pod adresem http://www.ollydbg.de/. Natywnym API dla programów Windows jest 32-bitowy Windows API, który programista Linuksa może traktować po prostu jako zbiór wszystkich udostępnionych biblioteki dostępne w /usr/lib. Doświadczony programista Linuksa może napisać program, który komunikuje się bezpośrednio z jądrem, na przykład za pomocą wywołań systemowych open() lub write(). Nie ma takiego szczęścia w systemie Windows. Każdy nowy dodatek Service Pack i wydanie systemu Windows NT zmienia interfejs jądra, a odpowiedni zestaw bibliotek (znanych jako biblioteki łączy dynamicznych [DLL]) jest dołączany do wydania, aby programy nadal działały. Biblioteki DLL umożliwiają procesowi wywołanie funkcji, która nie jest częścią własnego kodu wykonywalnego. Kod wykonywalny funkcji znajduje się w bibliotece DLL zawierającej jedną lub więcej funkcji, które są kompilowane, łączone i przechowywane oddzielnie od procesów, które z nich korzystają. Interfejs API systemu Windows jest zaimplementowany jako uporządkowany zestaw bibliotek DLL, więc każdy proces korzystający z interfejsu API Win32 używa dynamicznego łączenia. Daje to zespołowi jądra systemu Windows możliwość zmiany wewnętrznych interfejsów API lub dodania do nich złożonych nowych funkcji, jednocześnie zapewniając rozsądnie stabilny interfejs API dla programistów. W przeciwieństwie do tego, nie możesz dodać nowego argumentu do wywołania systemowego w jakimkolwiek wariancie Uniksa bez hordy programistów wywołujących faul. Jak każdy nowoczesny system operacyjny, Windows używa relokowalnego formatu pliku, który jest ładowany w czasie wykonywania, aby zapewnić funkcjonalność bibliotek współdzielonych. W Linuksie byłyby to pliki .so, ale w Windows są to pliki DLL. Podobnie jak .so jest plikiem ELF, DLL jest plikiem PE-COFF (zwanym również PE - przenośnym plikiem wykonywalnym). PE-COFF pochodził z formatu Unix COFF. Pliki PE są przenośne, ponieważ można je załadować na każdą 32-bitową platformę Windows; loader PE akceptuje ten format pliku. Plik PE zawiera tabelę importu i eksportu na początku pliku, która wskazuje zarówno jakie pliki PE musi znaleźć, jak i jakie funkcje wewnątrz tych plików potrzebuje. Eksport wskazuje, jakie funkcje zapewnia biblioteka DLL. Zaznacza również, gdzie w pliku, po załadowaniu do pamięci, znaleźć funkcje. Tablica importu zawiera listę wszystkich funkcji, których używa plik PE, które znajdują się w bibliotekach DLL, a także wymienia nazwę biblioteki DLL, w której znajduje się importowana funkcja. Większość plików PE można przenosić. Podobnie jak pliki ELF, plik PE składa się z różnych sekcji; sekcja .reloc może być użyta do przeniesienia biblioteki DLL w pamięci. Celem sekcji .reloc jest umożliwienie jednemu programowi załadowania dwóch bibliotek DLL, które zostały skompilowane do korzystania z tego samego miejsca w pamięci. W przeciwieństwie do Uniksa, domyślnym zachowaniem systemu Windows jest wyszukiwanie bibliotek DLL w bieżącym katalogu roboczym, zanim rozpocznie wyszukiwanie gdziekolwiek indziej. Daje to pewne możliwości uniknięcia ograniczeń Citrix lub Terminal Server z perspektywy hakera, ale z perspektywy programisty pozwala programiście aplikacji na dystrybucję wersji biblioteki DLL, która może różnić się od tej w katalogu głównym systemu (\winnt\system32) . Ten rodzaj problemu z wersjami jest czasami nazywany piekłem DLL. Użytkownicy będą musieli dostosować swoją zmienną środowiskową PATH i przenosić biblioteki DLL, aby nie kolidowały ze sobą podczas próby załadowania uszkodzonego programu. Ważną pierwszą rzeczą, którą należy się dowiedzieć o PE-COFF, jest względny adres wirtualny (RVA). RVA są używane do zmniejszenia ilości pracy, którą musi wykonać ładowarka PE. Funkcje można przenosić w dowolne miejsce wirtualnej przestrzeni adresowej; byłoby bardzo drogie, gdyby ładowarka PE musiała naprawiać każdy przemieszczalny przedmiot. Zauważysz, gdy nauczysz się Win32, którego Microsoft zwykle używa akronimu (RVA, AV [Access Violation], AD [Active Directory] i tak dalej) zamiast skracania samych terminów, jak to ma miejsce w systemie Unix (tmp, itp., vi, segfault). Każdy nowy dokument Microsoft wprowadza kilka tysięcy dodatkowych terminów i powiązanych z nimi akronimów. RVA jest

po prostu skrótem do powiedzenia „Każda biblioteka DLL jest ładowana do pamięci pod adresem podstawowym, a następnie dodajesz RVA do adresu podstawowego, aby coś znaleźć". Na przykład funkcja malloc() znajduje się w bibliotece DLL msvcrt.dll. Nagłówek w msvcrt.dll zawiera tabelę funkcji udostępnianych przez msvcrt.dll, tabelę eksportu. Tabela eksportu zawiera ciąg znaków z malloc i RVA (na przykład w 2000); po załadowaniu biblioteki DLL do pamięci, być może pod adresem 0x80000000, funkcję malloc można znaleźć, przechodząc do 0x80002000. Domyślna lokalizacja systemu Windows NT, do której ładowany jest plik .EXE, to 0x4000000. Może się to zmienić w zależności od pakietów językowych lub opcji kompilatora, ale jest to dość standardowe. Symbole plików PE-COFF dystrybuowanych przez firmę Microsoft są zwykle zawarte zewnętrznie. Możesz pobrać pakiety symboli dla każdej wersji swoich systemów operacyjnych z witryny sieci Web MSDN firmy Microsoft lub zdalnie użyć jego serwera symboli za pomocą WinDbg. OllyDbg nie obsługuje obecnie zdalnego serwera symboli. Aby uzyskać więcej informacji na temat PE-COFF, wyszukaj w witrynie internetowej firmy Microsoft hasło „PE-COFF". Na koniec pamiętaj, że podobnie jak kilka zepsutych Uniksów, Windows NT nie pozwoli ci usunąć aktualnie używanego pliku.

Sterta

Gdy biblioteka DLL zostanie załadowana, wywołuje funkcję inicjującą. Ta funkcja często konfiguruje własną stertę za pomocą HeapCreate() i przechowuje zmienną globalną jako wskaźnik do tej sterty, aby przyszłe operacje alokacji mogły jej używać zamiast sterty domyślnej. Większość bibliotek DLL ma sekcję .data w pamięci do przechowywania zmiennych globalnych i często można znaleźć przydatne wskaźniki do funkcji lub struktury danych przechowywane w tym obszarze. Ponieważ załadowanych jest wiele bibliotek DLL, istnieje wiele stert. Przy tak wielu stosach, które należy śledzić, ataki zepsucia stosu mogą stać się dość mylące. W Linuksie zazwyczaj może ulec uszkodzeniu jedna sterta, ale w systemie Windows może ulec uszkodzeniu kilka stert naraz, co znacznie komplikuje analizę sytuacji. Gdy użytkownik wywołuje malloc() w Win32, w rzeczywistości używa funkcji wyeksportowanej przez msvcrt.dll, która następnie wywołuje HeapAllocate() z prywatną stertą msvcrt.dll. Możesz ulec pokusie użycia funkcji HeapValidate() do analizy sytuacji uszkodzenia sterty, ale ta funkcja nie robi nic użytecznego. Zamieszanie zwykle pojawia się, gdy skończyłeś wykorzystywać przepełnienie sterty i chcesz wywołać niektóre funkcje API Win32 za pomocą kodu powłoki. Niektóre z twoich funkcji będą działać, a niektóre spowodują naruszenia dostępu w RtlHeapFree() lub RtlHeapAllocate, co może zakończyć proces, zanim będziesz miał szansę przejąć kontrolę. WinExec() i tym podobne są znane z tego, że nie pracują z uszkodzoną stertą. Każdy proces ma domyślną stertę. Domyślną stertę można znaleźć za pomocą GetDefaultHeap(), chociaż jest mało prawdopodobne, aby ta sterta została uszkodzona. Ważną rzeczą do zapamiętania jest to, że stosy mogą rosnąć w różnych segmentach. Na przykład, jeśli wyślesz wystarczającą ilość danych do usług IIS, zauważysz, że przydzielają one segmenty w wyższych zakresach pamięci i używają ich do przechowywania danych. Manipulowanie pamięcią w ten sposób może być użyteczną sztuczką, jeśli masz ograniczony zestaw znaków, którymi można nadpisać adres zwrotny, i jeśli chcesz uciec od adresu o małej ilości pamięci w domyślnych stertach. Z tego powodu wycieki pamięci w programach docelowych mogą stać się całkiem przydatne, ponieważ pozwalają wypełnić całą pamięć programu swoim kodem powłoki. Przepełnienie sterty w systemie Windows jest tak samo łatwe do napisania, jak w systemie Unix. Użyj tych samych podstawowych technik, aby je wykorzystać - jeśli będziesz ostrożny, możesz nawet wycisnąć więcej niż jeden zapis z przepełnienia sterty w systemie Windows, co znacznie ułatwia niezawodne wykorzystanie.

Wątkowanie

Wątkowanie umożliwia jednemu procesowi wykonywanie wielu czynności, dzieląc pojedynczą przestrzeń pamięci. Jądro systemu Windows udostępnia wycinki czasu procesora wątkom, a nie procesom. Linux robi rzeczy z modelem „lekkiego procesu", który jest dość słaby; tylko wtedy, gdy

Linux Native Threads zostanie zaimplementowany, Linux będzie miał stabilną podstawę wątków z resztą współczesnego świata systemów operacyjnych. Wątki po prostu nie są tak ważnym modelem programowania pod Linuksem z powodów, które staną się jasne, gdy wyjaśniona zostanie struktura bezpieczeństwa NT. Wątek jest powodem HRESULT. HRESULT, zasadniczo wartość całkowita, jest zwracana przez prawie wszystkie wywołania interfejsu Win32 API. HRESULT może być wartością błędu lub OK. Jeśli jest to wartość błędu, możesz uzyskać konkretny błąd za pomocą funkcji GetLastError(), która pobiera wartość z lokalnej pamięci wątku. Jeśli myślisz o modelu Uniksa, nie ma sposobu, aby odróżnić errno jednego wątku od drugiego. Win32 został zaprojektowany od podstaw jako model z gwintem. Windows nie ma fork() (używanego do tworzenia nowego procesu w Linuksie). Zamiast tego CreateProcess() stworzy nowy proces, który ma własną przestrzeń pamięci. Ten proces może dziedziczyć dowolne uchwyty, które jego rodzic oznaczył jako dziedziczne. Jednak rodzic musi następnie przekazać te uchwyty samemu dziecku lub pozwolić dziecku zgadnąć ich wartości (uchwyty są zazwyczaj małymi liczbami całkowitymi, jak uchwyty plików). Ponieważ prawie wszystkie przepełnienia występują w wątkach, atakujący nigdy nie zna prawidłowego adresu stosu. Oznacza to, że atakujący prawie zawsze używa triku w stylu powrotu do biblioteki DLL (chociaż używa dowolnej biblioteki DLL, nie tylko libc lub jej odpowiednika), aby uzyskać kontrolę nad wykonaniem.

Geniusz i idiotyzm rozproszonego modelu wspólnych obiektów i DCE-RPC

Distributed Common Object Model (DCOM), DCE-RPC, NT's Threading and Process Architecture oraz NT's Authentication Tokens są ze sobą połączone. Pomaga najpierw zrozumieć ogólną filozofię COM, aby zrozumieć, co odróżnia COM od jego uniksowych odpowiedników. Należy pamiętać, że pozycja Microsoftu w kwestii oprogramowania zawsze polegała na rozpowszechnianiu pakietów binarnych za pieniądze i budowaniu gospodarki, która to wspiera. Dlatego każda architektura oprogramowania firmy Microsoft obsługuje ten model. Możesz zbudować dość złożoną aplikację w całości, kupując moduły COM innych firm od różnych dostawców, wrzucając je do struktury katalogów, a następnie używając skryptu Visual Basic do łączenia ich ze sobą. Obiekty COM mogą być napisane w dowolnym języku, który obsługuje COM i bezproblemowo współdziałać. Większość dziwactw COM pojawia się jako naturalne decyzje projektowe; na przykład to, co jest liczbą całkowitą w C++, może nie być liczbą całkowitą w Visual Basic. Aby zagłębić się w COM, powinieneś spojrzeć na typowy plik języka opisu interfejsu (IDL). Użyjemy pliku DCOM IDL, który rozpoznasz później:

[uuid(e33c0cc4-0482-101a-bc0c-02608c6ba218),

version(1.0),

implicit_handle(handle_t rpc_binding)

] interface ???

{

typedef struct {

TYPE_2 element_1;

TYPE_3 element_2;

} TYPE_1;

…

short Function_00(

[in] long element_9,

[in] [unique] [string] wchar_t *element_10,

[in] [unique] TYPE_1 *element_11,

[in] [unique] TYPE_1 *element_12,

[in] [unique] TYPE_2 *element_13,

[in] long element_14,

[in] long element_15,

[out] [context_handle] void *element_16

);

To, co tutaj zdefiniowaliśmy, jest podobne do pliku nagłówkowego klasy C++. Mówi po prostu, że są to argumenty (i wartości zwracane) dla konkretnej funkcji w określonym interfejsie, zgodnie z definicją tego UUID. Wszystko, co musi być unikatowe - dowolna nazwa - jest identyfikatorem GUID w modelu COM. Ta 128-bitowa liczba ma być globalnie unikalna; to znaczy, że może być tylko jeden. Za każdym razem, gdy widzimy odniesienie do tego konkretnego UUID, wiemy, że mówimy o tym dokładnym interfejsie. Opisy interfejsów dla obiektów COM mogą być dowolnie złożone. Kompilator (i obsługa COM) dla języka ma stworzyć kawałek kodu, który może przekształcić się tak długo, jak określa go IDL, do formatu, w którym język wymaga, aby był reprezentowany. Tak samo jest ze znakami, tablicami, wskaźnikami przechowywanymi w tablicach, strukturami, które mają inne tablice i tak dalej. W praktyce można iść na skróty, aby utrzymać akceptowalną prędkość. Mówiąc, że long będzie 32 bity w kolejności little-endian, transformacja z C++ do innej reprezentacji obiektu COM C++ jest trywialna. Obiekt COM można wywołać na dwa sposoby: można go załadować bezpośrednio do przestrzeni procesu jako bibliotekę DLL lub uruchomić jako usługę (przez Menedżera sterowania usługami, specjalny proces działający jako SYSTEM). Uruchomienie serwera COM w innym procesie zapewnia, że Twój proces będzie stabilny i bezpieczniejszy, choć znacznie wolniej. Wywołania w trakcie procesu, które nie wymagają transformacji typów danych, są dosłownie tysiąc razy szybsze niż wywoływanie interfejsu COM na tej samej maszynie, ale w innym procesie. Przejście do tej samej maszyny jest zwykle co najmniej dziesięć razy szybsze niż przejście do maszyny w tej samej sieci. Dla Microsoftu ważne było to, że programiści mogli dokonać prostej zmiany w rejestrze lub zmienić jeden parametr w programie, a następnie ten program używałby innego procesu lub innej maszyny do wykonania tego samego wywołania. Na przykład spójrz na usługę AT w NT. Gdybyś miał napisać program do interakcji z AT i zaplanowania poleceń, mógłbyś wyszukać definicję interfejsu dla usługi AT, wykonać wywołanie DCOM, aby powiązać się z tym interfejsem, a następnie wywołać określoną procedurę na tym interfejsie. Oczywiście potrzebujesz pliku IDL, aby wiedzieć, jak przekształcić swoje argumenty, zanim wyślesz dane między swoim procesem a procesem usługi AT. Ta sama procedura działałaby, nawet jeśli proces byłby całkowicie na innym komputerze. W takim przypadku biblioteki DCOM połączyłyby się z maperem punktów końcowych komputera zdalnego (port TCP 135), a następnie zapytałyby go, gdzie nasłuchiwała usługa AT. Mapper punktów końcowych (sam jest usługą DCOM, ale taką, która jest zawsze na znanym porcie) odpowie „Usługa AT nasłuchuje na następujących nazwanych usługach RPC potoku, z którymi można się połączyć przez porty 445 lub 139. Nasłuchuje również na porcie TCP 1025 i porcie UDP 1034 dla połączeń DCE-RPC.” Wszystko to byłoby przejrzyste dla dewelopera. Teraz znasz geniusz DCE-RPC i DCOM. Możesz sprzedawać binarne pakiety DCOM lub po prostu postawić maszynę dostępną przez sieć z zainstalowanymi interfejsami DCOM i pozwolić programistom łączyć się z nimi z Visual Basic, C++ lub dowolnego innego języka obsługującego DCOM. Aby uzyskać dodatkową

szybkość, możesz załadować interfejsy bezpośrednio do procesu klienta jako bibliotekę DLL. Ten paradygmat jest podstawą prawie wszystkich funkcji, które czynią z Windows NT wyróżniającą się platformę serwerową. „Zaawansowani klienci", „Zdalne zarządzanie" i „Szybkie tworzenie aplikacji" to jedno i to samo — DCOM. Ale oczywiście jest to również idiotyzm DCE-RPC i DCOM. Możliwość zdalnego zarządzania przez jednego człowieka jest podatnością na zdalną lukę innego człowieka. Twoim celem jako hakera jest poznanie systemów docelowych lepiej niż ich administratorzy. Z DCOM jako złożoną, niemożliwą do zrozumienia podstawą dla każdego aspektu bezpieczeństwa systemu, nie jest to trudne. W następnych sekcjach omówiono kilka podstaw korzystania z DCE-RPC i DCOM.

Zwiad

Dwa przydatne narzędzia do podstawowego zdalnego rozpoznania DCE-RPC to SPIKE Dave'a Aitela (www.immunitysec.com/) oraz narzędzia DCE-RPC Todda Sabina (dostępne na stronie http://www.bindview.com/Services/razor/Utilities/). W tym przykładzie użyjemy narzędzia dcedump SPIKE, aby wyświetlić usługi DCE-RPC (znane również jako interfejsy DCOM) dostępne zdalnie, które są zarejestrowane w programie mapującym punkty końcowe. Jest to mniej więcej to samo, co wywołanie rpcdump -p w systemie Unix.

[dave@localhost dcedump]$ ./dcedump 192.168.1.108 | head -20

DCE-RPC tester.

TcpConnected

Entrynum=0

annotation=

uuid=4f82f460-0e21-11cf-909e-00805f48a135 , version=4

Executable on NT: inetinfo.exe

ncacn_np:\\WIN2KSRV[\PIPE\NNTPSVC]

Entrynum=1

annotation=

uuid=906b0ce0-c70b-1067-b317-00dd010662da , version=1

Executable on NT: msdtc.exe

ncalrpc[LRPC000001f4.00000001]

Entrynum=2

annotation=

uuid=906b0ce0-c70b-1067-b317-00dd010662da , version=1

Executable on NT: msdtc.exe

ncacn_ip_tcp:192.168.1.108[1025]

...

Jak widać, mamy tutaj trzy różne interfejsy i trzy różne sposoby łączenia się z nimi. Możemy dalej zbadać interfejs udostępniany przez program mapujący punkty końcowe za pomocą narzędzia do

identyfikatorów interfejsu (ifids) SPIKE. Podobnie możemy zbadać prawie każdy inny interfejs obsługujący protokół TCP (msdtc.exe jest jednym wyjątkiem).

```
[dave@localhost dcedump]$ ./ifids 192.168.1.108 135

DCE-RPC IFIDS by Dave Aitel.

Finds all the interfaces and versions listening on that TCP port

Tcp Connected

Found 11 entries

e1af8308-5d1f-11c9-91a4-08002b14a0fa v3.0

0b0a6584-9e0f-11cf-a3cf-00805f68cb1b v1.1

975201b0-59ca-11d0-a8d5-00a0c90d8051 v1.0

e60c73e6-88f9-11cf-9af1-0020af6e72f4 v2.0

99fcfec4-5260-101b-bbcb-00aa0021347a v0.0

b9e79e60-3d52-11ce-aaa1-00006901293f v0.2

412f241e-c12a-11ce-abff-0020af6e7a17 v0.2

00000136-0000-0000-c000-000000000046 v0.0

c6f3ee72-ce7e-11d1-b71e-00c04fc3111a v1.0

4d9f4ab8-7d1c-11cf-861e-0020af6e7c57 v0.0

000001a0-0000-0000-c000-000000000046 v0.0

Done
```

Teraz można je wprowadzić bezpośrednio do programu msrpcfuzz SPIKE, aby spróbować znaleźć przepełnienia w programie mapującym punkty końcowe lub w dowolnej innej usłudze TCP. Gdybyś miał IDL dla tych usług (niektóre z nich możesz uzyskać z projektów open source, takich jak Snort), mógłbyś pokierować analizą tych funkcji. W przeciwnym razie zostaniesz zredukowany do automatycznej lub ręcznej analizy binarnej. Jednym z programów, który może ci pomóc, jest Muddle, autorstwa Matta Chapmana. Program ten można znaleźć na stronie www.cse.unsw.edu.au/~matthewc/muddle/; automatycznie zdekoduje niektóre pliki wykonywalne, aby przekazać ci ich argumenty. Muddle wygenerował fragment IDL, który widziałeś wcześniej w tym rozdziale, który pobraliśmy z pliku usługi lokalizatora RPC. Firma Microsoft tunelowała protokół DCE-RPC przez prawie wszystko, co może znaleźć w swoich rękach. Od SMB do SOAP, jeśli możesz tunelować DCE-RPC przez to, masz włączone wszystkie narzędzia Microsoftu. W przykładach widać DCE-RPC przez nazwany interfejs potoku (ncacn_np), DCE-RPC przez lokalny interfejs RPC i DCE-RPC przez interfejs TCP. Nazwane interfejsy rur, TCP i UDP są dostępne zdalnie i powinny sprawić, że ślinka cieknie.

Eksploatacja

Istnieje tyle sposobów wykorzystania zdalnej usługi DCOM, ile jest wykorzystania zdalnej usługi SunRPC. Możesz wykonywać ataki w stylu popen() lub system(), próbować uzyskać dostęp do plików w systemie plików, znaleźć przepełnienie bufora lub podobne ataki, spróbować ominąć uwierzytelnianie lub cokolwiek innego, co możesz wymyślić, na co zdalny serwer może być podatny.

Najlepszym obecnie dostępnym publicznie narzędziem do zabawy z usługami RPC jest SPIKE. Jeśli jednak chcesz wykorzystać zdalne usługi DCE-RPC, będziesz musiał wykonać dużo pracy, duplikując ten protokół w wybranym przez siebie języku. CANVAS (www.immunitysec.com/CANVAS/) powiela DCE-RPC za pomocą Pythona. Na początku możesz pokusić się o użycie wewnętrznych interfejsów API firmy Microsoft do pracy nad wykorzystaniem DCE-RPC lub DCOM, ale na dłuższą metę Twoja niezdolność do bezpośredniego kontrolowania interfejsów API doprowadzi do tandetnych exploitów. Zdecydowanie używaj własnej lub implementacji protokołu open source, jeśli to możliwe.

Tokeny i personifikacja

Tokeny są dokładnie tym, na co brzmią - reprezentacją praw dostępu. W systemie Windows twoje prawa dostępu do takich rzeczy, jak pliki lub procesy nie są definiowane przez prosty zestaw uprawnień użytkownika/grupy/dowolnego, tak jak w systemie Linux. Zamiast tego używają elastycznego i niezwykle słabo poznanego mechanizmu, który opiera się na tokenach. W najmniejszym sensie token jest po prostu 32-bitową liczbą całkowitą, podobnie jak uchwyt pliku. Jądro NT utrzymuje wewnętrzną strukturę na procesu który wskazuje, co reprezentuje każdy token pod względem praw dostępu. Na przykład, gdy proces chce zainicjować inny proces, musi sprawdzić, czy może uzyskać dostęp do pliku, który chce zainicjować. Teraz sytuacja się komplikuje, ponieważ istnieje kilka rodzajów tokenów, a dwa tokeny mogą wpływać na każdą operację: token podstawowy i token bieżącego wątku. Proces otrzymał token podstawowy po uruchomieniu. Bieżący token wątku można uzyskać z innego procesu lub z funkcji LogonUser(). Funkcja LogonUser() wymaga podania nazwy użytkownika i hasła, a jeśli się powiedzie, zwraca nowy token. Możesz dołączyć dowolny dany token do bieżącego wątku za pomocą SetThreadToken(token_to_attach) i usunąć go za pomocą RevertToSelf(), w którym to momencie wątek powraca do tokenu podstawowego. Dla zabawy załaduj Sysinternals (http://www.microsoft.com/technet/sysinternals/) Process Explorer do procesu, a zobaczysz kilka rzeczy: Podstawowy token jest drukowany jako nazwa sera i możesz zobaczyć jeden lub więcej tokenów o różnych poziomach dostępu wymienionych w dolnym okienku. Uzyskanie tokena z innego procesu jest proste: jądro poda token dowolnego procesu, który jest dołączony do nazwanego potoku, który utworzyłeś, jeśli wywołasz funkcję ImpersonateNamedPipeClient(). Podobnie możesz podszywać się pod zdalnych klientów DCERPC lub dowolnego klienta, który podaje nazwę użytkownika i hasło. Na przykład, gdy użytkownik łączy się z uniksowym serwerem ftp, serwer ten działa jako root, więc może użyć setuid() do zmiany identyfikatora użytkownika na dowolnego użytkownika, jako którego uwierzytelnia się klient. W systemie Windows użytkownik wysyła nazwę użytkownika i hasło, a następnie serwer ftp wywołuje funkcję LogonUser(), która zwraca nowy token. Następnie tworzy nowy wątek, który wywołuje SetThreadToken(new_token). Gdy ten wątek zakończy obsługę klienta, wywołuje RevertToSelf() i dołącza do puli wątków lub wywołuje ExitThread() i znika. Pomyśl o tej procedurze jako o szansie dla hakera - w Uniksie, gdy wykorzystałeś serwer FTP z przepełnieniem bufora po uwierzytelnieniu, nie możesz zostać rootem ani żadnym innym użytkownikiem. W systemie Windows prawdopodobnie znajdziesz tokeny od wszystkich użytkowników, którzy niedawno się uwierzytelnili, czekając na Ciebie w pamięci aby chwytać je i używać. Oczywiście w wielu przypadkach sam serwer ftp będzie działał jako SYSTEM i możesz wywołać RevertToSelf(), aby uzyskać ten przywilej. Jedno powszechne nieporozumienie dotyczy CreateProcess(). Hakerzy uniksowi często wywołują execve("/bin/sh") jako część swojego kodu powłoki, ale w systemie Windows CreateProcess() używa podstawowego tokena jako tokena dla nowego procesu i używa bieżącego tokena wątku dla całego dostępu do pliku. Oznacza to, że jeśli bieżący token podstawowy ma niższy poziom dostępu niż token bieżącego wątku, nowy proces może nie być w stanie odczytać lub usunąć własnego pliku wykonywalnego. Dobrą ilustracją tego dziwactwa jest to, co dzieje się podczas ataku IIS. Zewnętrzne komponenty IIS działają wewnątrz procesów, których podstawowymi tokenami są IUSR lub IWAM, a nie SYSTEM. Jednak te procesy często mają wątki, które działają w nich jako SYSTEM. Gdy przepełnienie

daje hakerom kontrolę nad jednym z tych wątków i pobierają plik oraz funkcję CreateProcess(), uruchamiają się jako IUSR lub IWAM, ale właścicielem pliku jest SYSTEM. Jeśli kiedykolwiek znajdziesz się w takiej sytuacji, masz dwie opcje: możesz użyć DuplicateTokenEx() do wygenerowania nowego tokena podstawowego, który możesz przypisać do wywołania CreateProcessAsUser(), lub możesz wykonać całą swoją pracę z bieżącego wątku przez ładowanie biblioteki DLL bezpośrednio do pamięci lub za pomocą prostego kodu powłoki, który robi wszystko, czego potrzebujesz, z oryginalnego procesu.

Obsługa wyjątków pod Win32

W systemie Linux procedury obsługi wyjątków są zazwyczaj globalne; innymi słowy, na proces. Program obsługi wyjątków ustawia się za pomocą wywołania systemowego signal(), które jest wywoływane za każdym razem, gdy wystąpi wyjątek, taki jak segfault (lub w terminologii Windows, AV). W systemie Windows ten globalny program obsługi (w ntdll.dll) przechwytuje wszystkie wyjątki, a następnie wykonuje dość złożoną procedurę, aby określić, gdzie daje kontrolę. Ponieważ model programowania w systemie Windows NT jest skoncentrowany na wątkach, model obsługi wyjątków jest również skoncentrowany na wątkach. Proces cmd.exe ma dwa wątki. Blok danych drugiego wątku (który będzie w fs:[0] podczas wykonywania) ma wskaźnik do połączonej listy (łańcucha) struktur wyjątków. Pierwszym elementem tej struktury jest wskaźnik do następnego handlera. Drugim elementem tej struktury (Structured Exception Handler [SEH]) jest wskaźnik do funkcji. Wskaźnik do następnego modułu obsługi jest ustawiony na -1, co oznacza brak obsługi. Jednakże, jeśli pierwsza procedura obsługi nie zdecyduje się na obsługę danego wyjątku, zrobi to następny (jeśli taki istnieje) i tak dalej. Jeśli żaden program obsługi nie chce zaakceptować wyjątku, obsługuje go domyślna procedura obsługi wyjątków dla procesu. Zwykle skutkuje to zakończeniem procesu. Jako haker powinieneś teraz zobaczyć kilka sposobów na przejęcie kontroli nad tym systemem poprzez przepełnienie sterty lub podobne ataki, które pozwalają zapisać słowo w pamięci. Z pewnością możesz nadpisać wskaźnik do łańcucha SEH. Każdy proces w aplikacji Win32 ma dostarczony system operacyjny SEH. SEH jest odpowiedzialny za wyświetlanie okna błędu, które informuje użytkownika, że aplikacja została zakończona. Jeśli zdarzy się, że masz uruchomiony debuger, SEH daje możliwość debugowania aplikacji. Inną możliwością jest nadpisanie wskaźnika funkcji dla procedury obsługi na stosie lub nadpisanie domyślnej obsługi wyjątków. W systemie Windows XP masz inną opcję: Obsługa wyjątków wektorowych. Zasadniczo jest to po prostu kolejna połączona lista, którą najpierw sprawdza kod obsługi wyjątków w ntdll.dll. Więc teraz masz zmienną globalną, która jest wywoływana przy każdym wyjątku - idealna do nadpisywania.

Debugowanie systemu Windows

Masz zasadniczo trzy opcje debugowania systemu Windows: łańcuch narzędzi Microsoft, WinDbg; debugger jądra, SoftICE; lub OllyDbg. Możesz także użyć programu Visual Studio, jeśli masz na to ochotę. Z tych opcji SoftICE jest prawdopodobnie jedną z najstarszych i najpotężniejszych. SoftICE posiadał język makr i mógł debugować przestrzeń jądra. Wadą SoftICE było to, że może być prawie niemożliwa do zainstalowania, a GUI jest nieco staroświecki. Jego głównym zastosowaniem jest debugowanie nowych sterowników urządzeń. Przez długi czas był to jedyny wybór dla hakera, dlatego dostępnych jest kilka dobrych tekstów, jak z niego korzystać. Podczas debugowania jądra, SoftICE ustawia wszystkie strony na zapisywalne; bądź świadomy tego faktu, jeśli przepełnienie jądra, z którym pracujesz, wydaje się działać tylko przy włączonym SoftICE. WinDbg można skonfigurować do debugowania jądra - chociaż wymaga kabla szeregowego i innego komputera - ale może być również bardzo dobry do debugowania przepełnienia przestrzeni użytkownika. WinDbg ma prymitywny język, ale interfejs użytkownika jest okropny - prawie niemożliwy do szybkiego i dokładnego użycia. Niemniej jednak, ponieważ Microsoft używa tego debugera, ma kilka fajnych zaawansowanych funkcji, takich

jak automatyczny dostęp do serwera symboli firmy Microsoft. CDB, odpowiednik WinDbg w wierszu poleceń, jest niezwykle elastyczny i może być preferowany dla osób uzależnionych od wiersza poleceń. Tak jak SPIKE to najlepszy fuzzer, jaki kiedykolwiek stworzono, tak OllyDbg jest najlepszym debuggerem, jaki kiedykolwiek stworzono. Obsługuje niesamowite funkcje, takie jak run-trace (które pozwalają wykonywać wsteczne) wyszukiwanie pamięci, punkty przerwania pamięci (możesz na przykład ustawić przerwę za każdym razem, gdy ktoś uzyskuje dostęp do czegokolwiek w globalnej przestrzeni danych MSVCRT.DLL), inteligentne okna danych, asembler, patcher plików - w zasadzie wszystko, czego potrzebujesz. Jeśli OllyDbg nie obsługuje czegoś, czego potrzebujesz, możesz wysłać e-maila do autora, a następna wersja prawdopodobnie to zrobi. Poświęć trochę czasu na dołączanie do procesów za pomocą OllyDbg, a następnie fuzzowanie ich za pomocą SPIKE i analizowanie ich wyjątków. Dzięki temu szybko zapoznasz się z doskonałym graficznym interfejsem użytkownika OllyDbg.

Błędy w Win32

W Win32 jest wiele błędów, a wiele z nich jest nieudokumentowanych i boleśnie odkrytych przez ludzi piszących szelkod. Na przykład LoadLibraryA(), która ładuje bibliotekę DLL do pamięci, nie powiedzie się, jeśli w PATH znajduje się kropka, a maszyna nie została załatana dla tego konkretnego błędu. Procedury WinSock zakończą się niepowodzeniem, jeśli stos nie jest wyrównany do słów. Różne inne interfejsy API są słabo udokumentowane w MSDN, jeśli w ogóle. Najważniejsze jest to, że gdy twój kod powłoki nie działa, przyczyną może być prawdopodobnie błąd w systemie Windows i być może będziesz musiał po prostu go obejść.

Pisanie kodu powłoki systemu Windows

Pisanie niezawodnego szelkodu Windows przez długi czas było nieco tajną sprawą. Problem polega na tym, że w przeciwieństwie do szelkodu uniksowego, nie masz wywołań systemowych ze znanym API. Zamiast tego proces załadował wskaźniki funkcji do funkcji zewnętrznych, takich jak CreateProcess() lub ReadFile(), do różnych miejsc w pamięci. Ale ty, napastnik, nie wiesz, gdzie w pamięci się one znajdują. Wczesny kod szelfowy zakładał po prostu, że znajdują się w określonym miejscu lub domyślał się, że znajdują się w jednym z kilku miejsc. Oznacza to jednak, że za każdym razem, gdy tworzysz exploit, musisz go wersjonować w kilku różnych dodatkach serwisowych lub plikach wykonywalnych. Sztuczka do pisania niezawodnego i wielokrotnego użytku szelkodu polega na tym, że Windows przechowuje wskaźnik do bloku środowiska procesu w znanej lokalizacji: FS:[0x30]. To plus 0xc to wskaźnik listy modułów kolejności ładowania. Teraz masz połączoną listę modułów, przez które możesz przejść w poszukiwaniu kernel32.dll. Z tego możesz znaleźć LoadLibraryA() i GetProcAddress(), które pozwolą Ci załadować potrzebne biblioteki DLL i znaleźć adresy innych potrzebnych funkcji. Będziesz chciał wrócić i ponownie przeczytać dokument PE-COFF z shelcodu Microsoftu, aby to zrobić. Ta technika prowadzi do dużego shelcodu ze względu na jego złożoność. To powiedziawszy, w ostatnich latach ewoluowało kilka technik, aby je zmniejszyć, w tym innowacyjne metody haszowania. W artykule opublikowanym w 2005 r. Dafydd Stuttard z NGS udokumentował 191-bajtowy shellcode wiążący powłokę - bez bajtów zerowych - który wykorzystuje kilka sprytnych sztuczek, aby zmniejszyć kod, w tym użycie 8-bitowego skrótu wymaganych nazw funkcji. Jest oczywiście inny sposób. Różni chińscy hakerzy pisali szelkod, który przeszukuje pamięć dla kernel32, ustawiając procedurę obsługi wyjątków. Zobacz różne exploity NSFOCUS dla tej techniki zastosowanej w praktyce przeciwko IIS. Nawet ten szelkod może być dość duży. Dlatego CANVAS używa oddzielnego szelkodu, który składa się z 150 bajtów zakodowanych przy użyciu dzielonego kodera addytywnego CANVAS (podobnego do kodera/dekodera XOR, ale przy użyciu addl zamiast xorl), który po prostu wykorzystuje obsługę wyjątków do przeszukiwania całej pamięci procesu dla innego zestawu shellcode poprzedzony 8

bajtami wartości tagu. Ten szelkod okazał się wysoce niezawodny, a ponieważ możesz umieścić swój główny ładunek w dowolnym miejscu pamięci, nie musisz się martwić ograniczeniami miejsca.

Przewodnik hakera po API Win32

VirtualProtect() - ustawia kontrolę dostępu do strony pamięci. Przydatne do zmiany segmentów .text na +w, aby móc modyfikować funkcje. SetDefaultExceptionHandler — zdemontuj to, aby znaleźć globalną lokalizację programu obsługi wyjątków dla danego dodatku Service Pack. TlsSetValue()/TlsGetValue() — Lokalna pamięć masowa wątków to przestrzeń, której każdy wątek może używać do przechowywania zmiennych specyficznych dla wątku (innych niż stos lub sterta). Czasami znajdują się tutaj cenne wskazówki, które twój szelkod może chcieć zniszczyć. WSASocket() - wywołanie WSASocket() zamiast socket() konfiguruje gniazdo, którego można używać bezpośrednio jako standardowe wejście lub standardowe wyjście. Ta technika może być użyta do stworzenia mniejszego szelkodu, jeśli używasz szelkodu, który odradza cmd.exe. (Problem w uchwytach gniazd utworzonych za pomocą funkcji socket() dotyczy atrybutu SO_OPENTYPE).

Wniosek

Poznałeś podstawowe różnice między eksploatacją w systemach Linux/Unix i Windows. Te same koncepcje wysokiego poziomu, takie jak wywołania systemowe i pamięć procesów, są obecne w systemie Windows, ale z punktu widzenia hakera implementacja jest zupełnie inna. Uzbrojony w swoją wiedzę na temat eksploatacji systemu Windows, będziesz mógł przejść do kolejnych części, które szczegółowo omawiają hakowanie systemu Windows.