

0x600

PROTIOPATŘENÍ

Sekrety žabičky druhu pralesnička strašná (*Phyllobates terribilis*) z tropické Ameriky obsahují extrémně účinný jed – jedna žabička vylučuje tolik jedu, že by to stačilo na otrávení deseti dospělých lidí. Jedinou příčinou, proč tato žabička má tak překvapivě mocnou ochranu, je jistý druh hada, který ji požívá a vyvinul si postupně vůči jejímu jedu resistenci.

Obranná reakce žab je nasnadě – vyvíjí se u nich stále silnější a silnější jed. Jedním z důsledků této koevoluce je to, že žáby jsou v bezpečí proti všem ostatním predátorům. Tento typ koevoluce probíhá také u hackerů. Jejich exploitační techniky jsou známé už léta, takže je logické, že se vyvíjejí i obranná protiopatření. Hackeři samozřejmě reagují tím, že hledají způsoby, jak tyto obrany obcházet a rozvracet, což zase vede k vytvoření nových obranných technik.

Tento cyklus inovací je ve skutečnosti docela blahodárný. I když vám viry a červi mohou způsobit dost trablů a přerušit vaše obchodní aktivity, takže utrpíte značné ztráty, nutí vás k reakci, která daný problém napraví. Červi se replikují tím, že exploitují existující slabá místa ve vadném softwaru. Často jsou tyto závady neodhaleny celé roky, ale relativně benigní červi, jako jsou CodeRed nebo Sasser, si vynutí, aby se takové závady opravily. Je to úplně stejné jako u planých neštovic u lidí – je lepší přetrpět menší vzplanutí nákazy v raném dětství, než se nakazit o mnoho let později, kdy to může mít velmi závažné následky.

Kdyby nebylo internetových červů, kteří přitáhli pozornost veřejnosti na chyby v bezpečnosti, mohly by zůstat neopraveny, takže byste byli pořád zranitelní a mohli podlehnout útoku někoho s mnohem zlovolnějšími úmysly, než je pouhá replikace. V tomto ohledu mohou červi a viry, pokud se na to díváme z dlouhodobého hlediska, skutečně posílit bezpečnost. Existují ovšem mnohem aktivnější způsoby, jak se dá posílit bezpečnost. K dispozici jsou defenzivní protiopatření, která se snaží anulovat účinek útoku, nebo působit preventivně, aby k útoku nemohlo vůbec dojít. Protiopatření je dost teoretický pojem, který neříká nic konkrétního – může se jím myslet nějaký produkt z oblasti bezpečnosti, nějaká sada zásad, program, nebo prostě pouze pozorný systémový administrátor. Defenzivní protiopatření se dají rozdělit do dvou skupin: ta, která se pokoušejí detekovat útok, a ta, jež se pokoušejí ochránit zranitelné místo.

0x610 Detekující protiopatření

První skupina protiopatření se pokouší detekovat vniknutí a nějak na ně reagovat. Detekčním procesem může být leccos – od administrátora, který pročítá protokolovací soubory, až k programu, který odposlouchává síť. Reakce může znamenat to, že se automaticky zničí připojení nebo proces, nebo pouze to, že administrátor pozorně prohlíží všechno, co se objeví na konzole stroje.

Pokud jste na pozici systémového administrátora, tak exploits, o nichž víte, nejsou ani zdaleka tak nebezpečné jako exploits, o nichž nevíte. Čím dříve se vniknutí detekuje, tím dříve se s ním dá něco udělat, a je pravděpodobnější, že jej bude možné zkrotit. Znepokojovat by vás měla především vniknutí, která nebyla odhalena celé měsíce.

Detekce vniknutí spočívá v tom, že se předjímá, co asi útočící hacker hodlá udělat. Jestliže to dobře odhadnete, budete vědět, co a kde máte hledat. Detekující protiopatření mohou hledat vzory útoků v souborech protokolů, v síťových paketech, nebo dokonce v paměti programu. Poté, co se vniknutí detekovalo, může být hacker vypuzen ze systému, je možné napravit škody napáchané v souborech tím, že budou obnoveny ze zálohy, a dají se identifikovat a opravit exploitovaná zranitelná místa. Detekující protiopatření jsou v elektronickém světě poměrně mocná, když máme k dispozici výbavu pro zálohování a obnovu ze zálohy.

Útočník tedy musí počítat s tím, že proti všemu, co udělá, mohou působit detekční protiopatření. Protože ale detekce nemusí být vždy okamžitá, existuje několik scénářů "rozbij, a rychle seber všechno, co se dá", kdy útočník nemusí brát detekci v úvahu. I v těchto scénářích je ovšem lepší, když se nemusí hned prchat. Jednou z nejceněnějších deviz hackera je utajení. Exploitujete-li nějaký zranitelný program, takže získáte přístup k shellu roota, znamená to, že na daném systému si můžete dělat cokoliv, co se vám zlíbí. Pokud si ale dáte dobrý pozor a vaše vniknutí nebude detekováno, znamená to navíc, že nikdo neví, že tam jste. A teprve kombinace "můžu všechno" a neviditelnost činí hackera nebezpečným. Z dobré skryše se dají nenápadně na síti odposlouchat data a hesla, lézt zadními vrátky do programů, a dají se připravovat další útoky na jiných hostitelích. Pokud chcete zůstat skrytí, musíte předpokládat existenci nějakých detekčních metod. Víte-li, co hledají a kde šňourají, budete se moci vyhnout exploitačním vzorům, které by vás odhalily, nebo můžete napodobovat nějaké platné vzory. Koevoluční cyklus mezi skrýváním a detekcí je poháněn přemýšlením o věcech, na které druhá strana vůbec nepomyslela.

0x620 Systémoví démoni

Aby naše diskuse o exploitaci protiopatření a metodách obcházení detekce mohla být realistická, potřebujeme především nějaký realistický cíl, který bychom mohli exploitovat. Naším cílem bude nějaký vzdálený serverový program, který akceptuje příchozí připojení. V Unixu jsou takovými programy obvykle systémoví démoni. Démon je program, který běží v pozadí a je jistým způsobem oddělen od řídicího terminálu. Termín démon (daemon) poprvé použili hackeři MIT v šedesátých letech dvacátého století. Odkazuje se na démona z myšlenkového experimentu, který v roce 1867 podnikl fyzik James Maxwell. V tomto experimentu je Maxwellův démon bytostí obdařenou

nadpřirozenou schopností bez námahy řešit obtížné úlohy; zde konkrétně šlo o úlohu oddělit pomalejší a rychlejší molekuly. Cílem experimentu bylo teoreticky zkoumat, zdali je možné porušit druhý zákon termodynamiky. Obdobnými schopnostmi jsou vybaveni i systémoví démoni v Linuxu – neúnavně provádějí úlohy jako poskytování služby SSH a udržování systémových protokolů. Programy, které mají charakter démona, obvykle končí na písmeno d, které vyznačuje, že jde o démona, například sshd nebo syslogd.

Když do kódu `tinyweb.c` z kapitoly 0x400 přidáme několik drobností, uděláme z něho realističtějšího systémového démona. Nový kód volá funkci `daemon()`, která zplodí nový proces v pozadí. Tuto funkci v Linuxu používají mnohé procesy systémových démonů. A zde je manuálová stránka této funkce.

DAEMON(3) Linux Programmer's Manual DAEMON(3)

NÁZEV

`daemon` – běží v pozadí

SYNOPSIS

```
#include <unistd.h>
int daemon(int nochdir, int noclose);
```

POPIS

Funkce `daemon()` je určena programům, které se chtějí oddělit od řídicího terminálu a běžet v pozadí jako systémoví démoni.

Pokud je argument `nochdir` nulový, `daemon()` změní aktuální pracovní adresář na `root ("/")`, jinak ne.

Pokud je argument `noclose` nulový, `daemon()` přesměruje standardní vstup, standardní výstup a standardní chybový výstup na `/dev/null`,

NÁVRATOVÁ HODNOTA

(Tato funkce rozdvouje, a pokud `fork()` uspěje, udělá rodičovský proces `_exit(0)`, takže další chyby vidí pouze dceřiný proces.) Při úspěchu se vrátí nula. Dojde-li k nějaké chybě, `daemon()` vrátí `-1` a nastaví globální proměnnou `errno` na jednu z chyb, které jsou specifikované pro knihovní funkce `fork(2)` a `setsid(2)`.

Protože systémoví démoni běží odděleně od řídicího terminálu, kód nového démona `tinyweb` zapisuje do nějakého protokolovacího souboru. Protože systémoví démoni běží nezávisle na řídicím terminálu, jsou obvykle řízeni pomocí signálů. Nový program `tinyweb` v podobě démona potřebuje zachytit ukončovací signál, aby mohl hladce skončit, pokud bude zabit (signálem `Kill`).

0x621 *Signály letem-světlem*

Signály v Unixu poskytují metodu pro komunikaci mezi jednotlivými procesy. Když proces obdrží signál, operační systém přeruší tok jeho vykonávání, aby se mohl zavolat zpracovatel signálu. Signály se identifikují čísly, přičemž každý signál má svého výchozího zpracovatele. Pokud například na řídícím terminálu programu stisknete CTRL+C, odešle se přerušovací signál (interrupt signal), který má výchozího zpracovatele signálu, jenž ukončí program. To vám umožňuje přerušit běh programu i v situaci, když se zasekl v nekonečném cyklu.

Pomocí funkce `signal()` se dají zaregistrovat vlastní zpracovatelé signálů. V následující ukázce kódu se pro jisté signály registrují zpracovatelé signálů; kód `main` obsahuje nekonečný cyklus.

```
signal_example.c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

/* Několik definic signálů z signal.h opatřených popisky
 * #define SIGHUP      1 Hangup
 * #define SIGINT      2 Interrupt (Ctrl-C)
 * #define SIGQUIT     3 Quit (Ctrl-\)
 * #define SIGILL      4 Neplatná instrukce
 * #define SIGTRAP     5 Trace/breakpoint trap
 * #define SIGABRT     6 Proces násilně ukončen
 * #define SIGBUS      7 Chyba Bus
 * #define SIGFPE      8 Chyba pohyblivé řádové čárky
 * #define SIGKILL     9 Kill
 * #define SIGUSR1    10 Uživatelsky definovaný signál 1
 * #define SIGSEGV    11 Selhání segmentace (Segmentation fault)
 * #define SIGUSR2    12 Uživatelsky definovaný signál 2
 * #define SIGPIPE    13 Zápis do trubice (pipe), ale žádná nečte
 * #define SIGALRM    14 Nastaveno odpočítávání funkcí alarm()
 * #define SIGTERM    15 Ukončení (odeslán příkazem kill)
 * #define SIGCHLD    17 Signál dceřiného procesu
 * #define SIGCONT    18 Pokračovat, pokud byl zastaven
 * #define SIGSTOP    19 Stop (pozastavit vykonávání)
 * #define SIGTSTP    20 Terminal stop [suspend] (Ctrl-Z)
 * #define SIGTTIN    21 Proces v pozadí se pokouší číst standardní vstup
 * #define SIGTTOU    22 Proces v pozadí se pokouší číst standardní výstup
 */

/* Zpracovatel signálů */
void signal_handler(int signal) {
```

```

printf("Caught signal %d\t", signal);
if (signal == SIGTSTP)
    printf("SIGTSTP (Ctrl-Z)");
else if (signal == SIGQUIT)
    printf("SIGQUIT (Ctrl-\\)");
else if (signal == SIGUSR1)
    printf("SIGUSR1");
else if (signal == SIGUSR2)
    printf("SIGUSR2");
printf("\n");
}

void sigint_handler(int x) {
    printf("Caught a Ctrl-C (SIGINT) in a separate handler\nExiting.\n");
    exit(0);
}

int main() {
    /* Registrace zpracovatelů signálů */
    signal(SIGQUIT, signal_handler); // Určí signal_handler() jako
    signal(SIGTSTP, signal_handler); // zpracovatele signálů těchto
    signal(SIGUSR1, signal_handler); // signálů.
    signal(SIGUSR2, signal_handler);

    signal(SIGINT, sigint_handler); // Nastaví sigint_handler() pro SIGINT.
    while(1) {}                     // Nekonečný cyklus.
}

```

Když se tento program zkompiluje a spustí, zaregistrují se zpracovatelé signálů a program vstoupí do nekonečného cyklu. Přestože se program zasekne v nekonečném cyklu, příchozí signály přerušují jeho vykonávání a zavolají se zaregistrovaní zpracovatelé signálů. Ve výstupu níže se používají ty signály, které lze spustit z řídícího terminálu. Když funkce `signal_handler()` skončí, vrátí vykonávání zpět do přerušenoého cyklu, zatímco funkce `sigint_handler()` program ukončí.

```

reader@hacking:~/booksrc $ gcc -o signal_example signal_example.c
reader@hacking:~/booksrc $ ./signal_example
Caught signal 20      SIGTSTP (Ctrl-Z)
Caught signal 3 SIGQUIT (Ctrl-\\)
Caught a Ctrl-C (SIGINT) in a separate handler
Exiting.
reader@hacking:~/booksrc $

```

Konkrétní signály se dají procesu odesílat příkazem `kill`. Tento příkaz standardně odešle procesu ukončující signál (`SIGTERM`). Uvede-li se volba `-l` příkazového řádku, `kill` vypíše seznam všech možných signálů. Ve výstupu níže jsou programu `signal_example`, který se vykonává v jiném terminálu, odeslány signály `SIGUSR1` a `SIGUSR2`.

```
reader@hacking:~/booksrc $ kill -l
1) SIGHUP          2) SIGINT          3) SIGQUIT         4) SIGILL
5) SIGTRAP         6) SIGABRT        7) SIGBUS          8) SIGFPE
9) SIGKILL         10) SIGUSR1       11) SIGSEGV         12) SIGUSR2
13) SIGPIPE        14) SIGALRM       15) SIGTERM         16) SIGSTKFLT
17) SIGCHLD        18) SIGCONT       19) SIGSTOP         20) SIGTSTP
21) SIGTTIN        22) SIGTTOU       23) SIGURG          24) SIGXCPU
25) SIGXFSZ        26) SIGVTALRM     27) SIGPROF         28) SIGWINCH
29) SIGIO          30) SIGPWR        31) SIGSYS          34) SIGRTMIN
35) SIGRTMIN+1     36) SIGRTMIN+2    37) SIGRTMIN+3     38) SIGRTMIN+4
39) SIGRTMIN+5     40) SIGRTMIN+6    41) SIGRTMIN+7     42) SIGRTMIN+8
43) SIGRTMIN+9     44) SIGRTMIN+10   45) SIGRTMIN+11    46) SIGRTMIN+12
47) SIGRTMIN+13    48) SIGRTMIN+14   49) SIGRTMIN+15    50) SIGRTMAX-14
51) SIGRTMAX-13    52) SIGRTMAX-12   53) SIGRTMAX-11    54) SIGRTMAX-10
55) SIGRTMAX-9     56) SIGRTMAX-8    57) SIGRTMAX-7     58) SIGRTMAX-6
59) SIGRTMAX-5     60) SIGRTMAX-4    61) SIGRTMAX-3     62) SIGRTMAX-2
63) SIGRTMAX-1     64) SIGRTMAX

reader@hacking:~/booksrc $ ps a | grep signal_example
24491 pts/3    R+      0:17 ./signal_example
24512 pts/1    S+      0:00 grep signal_example
reader@hacking:~/booksrc $ kill -10 24491
reader@hacking:~/booksrc $ kill -12 24491
reader@hacking:~/booksrc $ kill -9 24491
reader@hacking:~/booksrc $
```

Nakonec se pomocí `kill -9` odešle signál `SIGKILL`. Protože není možné změnit zpracovatele tohoto signálu, příkaz `kill -9` lze vždy použít pro zabití procesu. V tom druhém okně terminálu, kde běží program `signal_example`, je vidět, které signály byly zachyceny, a že proces byl zabit.

```
reader@hacking:~/booksrc $ ./signal_example
Caught signal 10      SIGUSR1
Caught signal 12      SIGUSR2
Killed
reader@hacking:~/booksrc $
```

Ačkoliv signály samy o sobě jsou velmi prosté, komunikací mezi jednotlivými procesy mohou rychle vzniknout komplexní webové závislosti. V našem novém démonovi `tinyweb` se ovšem signály používají pouze pro hladké ukončení procesu, takže jejich implementace byla jednoduchá.

0x622 *Démon tinyweb*

Tato novější verze programu tinyweb je systémovým démonem, který běží v pozadí bez řídicího terminálu. Svůj výstup s časovými známkami zapisuje do protokolovacího souboru a naslouchá signálu pro ukončení (SIGTERM), takže se může hladce shodit, když je zabit.

Ačkoliv změny v programu jsou opravdu velmi drobné, poskytují mnohem realističtější cíl exploitace. Nové části kódu jsou v následujícím výpisu zvýrazněny tučně.

tinywebd.c

```
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <signal.h>
#include "hacking.h"
#include "hacking-network.h"

#define PORT 80 // Port, na který se budou
                // uživatelé připojovat.

#define WEBROOT "./webroot" // Kořenový adresář webového serveru
#define LOGFILE "/var/log/tinywebd.log" // Název protokolovacího souboru

int logfd, sockfd; // Globální deskriptory pro
                  // protokolovací soubor a socketový
                  // soubor

void handle_connection(int, struct sockaddr_in *, int);
int get_file_size(int); // Vrátí velikost souboru otevřeného
                        // souborového deskriptoru

void timestamp(int); // Zapiše časovou známku
                    // do otevřeného souborového
                    // deskriptoru

// Tato funkce se zavolá, když je proces zabit (killed).
void handle_shutdown(int signal) {
    timestamp(logfd);
    write(logfd, "Shutting down.\n", 16);
    close(logfd);
    close(sockfd);
    exit(0);
}
```

```

int main(void) {
    int new_sockfd, yes=1;
    struct sockaddr_in host_addr, client_addr; // Informace o mé adrese
    socklen_t sin_size;

    logfd = open(LOGFILE, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
    if(logfd == -1)
        fatal("opening log file");

    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
        fatal("in socket");

    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
        fatal("setting socket option SO_REUSEADDR");

    printf("Starting tiny web daemon.\n");
    if(daemon(1, 0) == -1) // Rozdvojení na proces démona v pozadí.
        fatal("forking to daemon process");

    signal(SIGTERM, handle_shutdown); // Když je zabit, zavolá
                                     // handle_shutdown.
    signal(SIGINT, handle_shutdown); // Když je přerušen, zavolá
                                     // handle_shutdown.

    timestamp(logfd);
    write(logfd, "Starting up.\n", 15);
    host_addr.sin_family = AF_INET; // Bajtové pořadí hostitele
    host_addr.sin_port = htons(PORT); // síťové bajtové pořadí short
    host_addr.sin_addr.s_addr = INADDR_ANY; // Automaticky vyplní
                                     // mou IP adresou.
    memset(&(host_addr.sin_zero), '\0', 8); // Vynuluje zbytek struktury.

    if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct
                                     sockaddr)) == -1)
        fatal("binding to socket");
    if (listen(sockfd, 20) == -1)
        fatal("listening on socket");

    while(1) { // Cyklus akceptace příchozích připojení.
        sin_size = sizeof(struct sockaddr_in);
        new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
        if(new_sockfd == -1)

```



```

        fatal("accepting connection");
        handle_connection(new_sockfd, &client_addr, logfd);
    }
    return 0;
}

/* Funkce zpracuje připojení na předaném socketu z předané klientské adresy
 * a zaprotokoluje předaný souborový deskriptor. Připojení se zpracuje jako
 * webový požadavek a tato funkce odpoví přes připojený socket. Nakonec se na
 * konci funkce předaný socket uzavře.
 */
void handle_connection(int sockfd, struct sockaddr_in
                        *client_addr_ptr, int logfd) {
    unsigned char *ptr, request[500], resource[500], log_buffer[500];
    int fd, length;
    length = recv_line(sockfd, request);
    sprintf(log_buffer, "From %s:%d \"%s\"%t", inet_ntoa(client_addr_ptr->
        sin_addr), ntohs(client_addr_ptr->sin_port), request);
    ptr = strstr(request, " HTTP/");           // Hledá požadavek,
                                                // který vypadá jako platný.
    if(ptr == NULL) {                           // Tohle není platný HTTP.
        strcat(log_buffer, " NOT HTTP!\n");
    } else {
        *ptr = 0;                               // Ukončí buffer na konci URL.
        ptr = NULL;                             // Nastaví ptr na NULL (označí tím
                                                // neplatný požadavek).

        if(strncmp(request, "GET ", 4) == 0) // požadavek Get
            ptr = request+4;                  // ptr je URL.
        if(strncmp(request, "HEAD ", 5) == 0) // požadavek Head
            ptr = request+5;                  // ptr je URL.
        if(ptr == NULL) {                     // Tohle nebylo rozpoznáno
                                                // jako požadavek.
            strcat(log_buffer, " UNKNOWN REQUEST!\n");
        } else {                              // Platný požadavek, ptr
                                                // ukazuje na název zdroje

            if (ptr[strlen(ptr) - 1] == '/') // U zdroje končícího na '/',
                strcat(ptr, "index.html");   // přidá nakonec 'index.html'..
            strcpy(resource, WEBROOT);        // Zahájí zdroj cestou
                                                // webového kořene
            strcat(resource, ptr);             // a sloučí ji s cestou zdroje.
            fd = open(resource, O_RDONLY, 0); // Pokusí se otevřít soubor.
            if(fd == -1) {                   // Pokud se soubor nenašel

```

```

    strcat(log_buffer, " 404 Not Found\n");
    send_string(sockfd, "HTTP/1.0 404 NOT FOUND\r\n");
    send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
    send_string(sockfd, "<html><head><title>404 Not Found</title>
                        </head>");
    send_string(sockfd, "<body><h1>URL not found</h1></body>
                        </html>\r\n");
} else {                                     // Jinak soubor obslouží.
    strcat(log_buffer, " 200 OK\n");
    send_string(sockfd, "HTTP/1.0 200 OK\r\n");
    send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
    if(ptr == request + 4) {                 // Pak je tohle požadavek GET
        if( (length = get_file_size(fd)) == -1)
            fatal("getting resource file size");
        if( (ptr = (unsigned char *) malloc(length)) == NULL)
            fatal("allocating memory for reading resource");
        read(fd, ptr, length);               // Načte soubor do paměti.
        send(sockfd, ptr, length, 0);        // Odešle ho socketu.
        free(ptr);                           // Uvolní paměť souboru.
    }
    close(fd);                               // Uzavře soubor.
}                                             // Konec bloku if pro soubor nalezen/nenalezen.
}                                             // Konec bloku if platného požadavku.
}                                             // Konec bloku if platného HTTP.
timestamp(logfd);
length = strlen(log_buffer);
write(logfd, log_buffer, length);           // Zapiše do protokolu.
shutdown(sockfd, SHUT_RDWR);               // Graciézně socket uzavře.
}

```

```

/* Funkce přebírá otevřený souborový deskriptor a vrací
 * velikost sdruženého souboru. Při nezdaru vrací -1.
 */

```

```

int get_file_size(int fd) {
    struct stat stat_struct;
    if(fstat(fd, &stat_struct) == -1)
        return -1;
    return (int) stat_struct.st_size;
}

```

```

/* Funkce zapiše řetězec časové známky do otevřeného

```

```

* souborového deskriptoru, který byl do ní předán.
*/
void timestamp(fd) {
    time_t now;
    struct tm *time_struct;
    int length;
    char time_buffer[40];
    time(&now); // Získá počet sekund od začátku epochy.
    time_struct = localtime((const time_t *)&now); // Převede na strukturu tm.
    write(fd, time_buffer, length); // Zapíše řetězec časové
    // známky do protokolu.
}

```

Tento démon se rozdvojuje (forks) do pozadí, zapisuje do souboru protokolu časové známky, a pokud je zabit, hladce skončí. Deskriptor protokolovacího souboru a socket přijímající připojení jsou deklarovány jako globální, takže se dají hladce uzavřít funkcí `handle_shutdown()`. Tato funkce je připravena jako zpracovatel zpětného volání (callback) pro ukončovací (terminate) a přerušovací (interrupt) signál, což umožňuje programu hladce skončit, když je zabit příkazem `kill`.

Následující výstup ukazuje, jak se program zkompiloval, spustil a zabil. Povšimněte si, že protokolovací soubor obsahuje jak časové známky, tak i zprávu o shození, když program zachytil ukončovací signál a zavolal `handle_shutdown()`, aby skončil hladce.

```

reader@hacking:~/booksrc $ gcc -o tinywebd tinywebd.c
reader@hacking:~/booksrc $ sudo chown root ./tinywebd
reader@hacking:~/booksrc $ sudo chmod u+s ./tinywebd
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.

```

```

reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $ ps ax | grep tinywebd
25058 ?        Ss          0:00 ./tinywebd
25075 pts/3    R+          0:00 grep tinywebd
reader@hacking:~/booksrc $ kill 25058
reader@hacking:~/booksrc $ ps ax | grep tinywebd
25121 pts/3    R+          0:00 grep tinywebd
reader@hacking:~/booksrc $ cat /var/log/tinywebd.log
cat: /var/log/tinywebd.log: Permission denied
reader@hacking:~/booksrc $ sudo cat /var/log/tinywebd.log
07/22/2007 17:55:45> Starting up.
07/22/2007 17:57:00> From 127.0.0.1:38127 "HEAD / HTTP/1.0"      200 OK
07/22/2007 17:57:21> Shutting down.
reader@hacking:~/booksrc $

```

Tento program `tinywebd.c` obsluhuje HTTP obsah úplně stejně, jako původní program `tinyweb`, ovšem s tím rozdílem, že se chová jako systémový démon, je oddělen od řídicího terminálu a zapisuje do protokolovacího souboru. Oba programy jsou ovšem zranitelné stejnými exploity využívajícími přetečení. Tato exploitační je nicméně pouhý začátek. Když za cíl exploitační zvolíme tohoto nového démona, dozvíme se, jak se po vniknutí vyhnout detekci.

0x630 Nástroje našeho řemesla

Když nyní máme po ruce realistický cíl, přelezme barikádu, abychom se dostali na teritorium útočníka. Pro tento druh útoku jsou nepostradatelnými nástroji našeho řemesla exploitační skripty. Tak jako dokáže profesionál se šperhákem v ruce otevřít různé zámky než byste stihnuli mrknout okem, exploity obdobným způsobem otvírají mnohé dveře hackerovi. Prostřednictvím pečlivé a obratné manipulace s interními mechanismy je možné se zcela vyhnout bezpečnostním opatřením.

V předchozích kapitolách jsme psali kód exploitu v C a z příkazového řádku ručně zkoumali slabinu programů. Nežřetelnou hranici, která odděluje exploitační programy od exploitačních nástrojů, tvoří finalizace a opětovná konfigurovatelnost. Exploitační program je spíše puška než nástroj. Podobně jako puška i exploitační program se používá jednorázově a jeho uživatelské rozhraní je prostě něco podobného jako kohoutek u pušky, který stačí zmáčknout. Střelné zbraně i exploitační programy jsou finální výrobky, s nimiž mohou zacházet nekvalifikované osoby, což může mít nebezpečné následky. Oproti tomu exploitační nástroje nelze označit za finální výrobky. Nejsou také určeny pro použití jinými osobami. Protože hacker umí programovat, je zcela přirozené, že si vytváří své vlastní skripty, které poté používá jako pomůcky při exploitační. Tyto personalizované nástroje automatizují pracné úlohy a ulehčují experimenty. Podobně jako běžné nástroje ze života, i tyto nástroje se mohou využívat pro nejrozumnější potřeby a rozvíjet tak dovednosti uživatele.

0x631 *Nástroj pro exploitační tinywebd*

Pro démona `tinyweb` bychom rádi měli takový exploitační nástroj, který by nám umožnil experimentovat s jeho zranitelnostmi. Podobně jako při vývoji předchozích exploitů i zde nejprve prostřednictvím GDB zjistíme o daném zranitelném místě nějaké podrobnosti, například offsety. Offset k návratové adrese bude stejný jako v původním programu `tinyweb.c`. Program, který je démonem, nám ovšem předkládá dodatečnou výzvu. Volání `daemon()` rozdvojí proces, což znamená, že zbytek programu běží v dceřiném procesu, zatímco rodičovský proces skončí. V následujícím výstupu jsme nastavili bod přerušování hned za volání `daemon()`, nicméně debugger se k němu nikdy nedostane.

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q ./a.out
```

```
warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
```