

Základy bezpečného programování pod (nejen) pro OS Linux

Jiří Kosina

MFF UK Praha
Email: jikos@jikos.cz

Abstrakt: V přednášce budou demonstrovány nejběžnější typy bezpečnostních chyb, kterých se programátor může při programování (nejen) pod linuxem dopustit, se zaměřením na buffer overflow, chyby při formátování řetězců, race conditions, případně nejběžnější chyby při vývoji webovských aplikací (cgi, php – špatná validace vstupních dat).

Klíčová slova: Buffer overflow, race condition, format string.

1 Úvod

Asi není příliš nutné zdůrazňovat, jak je v dnešní době důležité psaní bezpečných aplikací – v době, kdy aplikace běží na Internetu, kde k nim má přístup takřka kdokoli, kdy se informace, které jsou spravovány těmito aplikacemi, stávají stále citlivějšími a kdy si takřka denně můžeme přečíst informace o strojích, které někdo proboural díky bezpečnostní chybě v aplikaci. Tato přednáška by měla sloužit jen jako velice jemný úvod do tvorby bezpečných aplikací, protože jde o problematiku velice rozsáhlou, a samozřejmě dosud ne zcela prozkoumanou.

2 Spouštění externích příkazů

Představme si následující program, který běží v systému s právy, která bychom, coby útočník, chtěli získat:

```
int main (void)
{
if (system ("mail nekdo@nekde.cz < /home/vypisy/vypis.txt"))
    perror ("system");
return (0);
}
```

Funkce `system()` spustí shell (typicky `/bin/sh`) s parametrem `-c`, za kterým mu předá své argumenty. V našem případě pak shell podle cesty, nastavené v proměnné prostředí `PATH`, zkouší najít program `mail`, který pak spustí. Pokud je tedy výše uvedený program zkompileovaný pod názvem `/usr/sbin/vuln1` a před jeho spuštěním si ještě vytvoříme ve svém adresáři skript s názvem `mail` s následujícím obsahem:

```
#!/bin/bash
/bin/sh
```

pak práva, která má program `/usr/sbin/vuln1`, získáme triviálně takto:

```
$ export PATH="."
$ /usr/sbin/vuln1
```

Výsledkem bude běžící shell se stejnými právy, s jakými se spouští program `vuln1`.

Jedním z možných řešení tohoto konkrétního problému je uvést celou cestu k programu, který pomocí `system()` spoušíme, a nespolehat se na nastavení proměnné `PATH`. Jednou z nevýhod tohoto přístupu je samozřejmě to, že se spoléháme na lokální konfiguraci systému, což u programů typu `ls`, `mail`, apod. není zásadní problém, ale ne vždy musíme u každého programu vědět, kde se v systému nachází (typicky jde o programy ručně instalované správcem do `/usr`, `/usr/local`, apod.). Další nevýhodou, a to poněkud podstatnější, je fakt, že i program, který používá plnou cestu k programu, je stále zranitelný pomocí obdobného útoku. Stačí si uvědomit existenci proměnné prostředí `IFS`, kterou používá shell (nejen) k parsování příkazové řádky. Obsah této proměnné určuje, které znaky mají být brány jako oddělovače polí. Defaultní nastavení této proměnné je mezera, tabulátor, atd. Co se však stane, budeme-li používat volání tvaru

```
system("/bin/mail ...");
```

pak pokud útočník provede něco na způsob:

```
$ export PATH="."
$ export IFS=$IFS/
$ /usr/sbin/vuln1
```

tak opět dosáhne spuštění skriptu s názvem `bin` v aktuálním adresáři (který útočník naplní pravděpodobně stejně, jako v předchozím případě skript s názvem `mail`).

Možných řešení těchto problémů je několik. Především je možné se vyhnout úplně používání volání `system()`, a místo něj si spuštění implementovat sám (pomocí `fork()`, `exec()`, přesměrování, které je při použití `system()` jednoduché (protože ho za nás řeší shell) řešit sami pomocí `dup()`, atd.

Dalším řešením je před voláním funkce `system()` si nastavit takové hodnoty proměnných prostředí, jaké chceme, a které jsou bezpečné. Například

```
clearenv();
setenv ("IFS", "\n\t",1);
setenv ("PATH", "/bin:/usr/bin:/usr/local/bin", 1);
system ("mail nekdo@nekde.cz < soubor");
```

3 Buffer overflow

Buffer overflow je souhrnný název pro chyby, které umožňují změnit některou část paměti programu, přestože programátor neměl původně v úmyslu nic takového dovolit. Myšlenkově lze tyto chyby rozdělit podle typu paměti, ve které k přepisu dojde, na *stack overflow* a *heap overflow*, přičemž na zneužití každé z nich musí útočník použít poněkud jiné techniky.

Stack (česky zásobník) je název pro paměť, kterou používají vzájemně se volající funkce k předávání parametrů (a dalších pomocných hodnot), a kompilátory do ní často umísťují lokální a statické proměnné. *Heap* (česky halda) je název pro paměť, do které kompilátor umísťuje globální a dynamická data (z této oblasti například vrací funkce `malloc()` kusy paměti).

Důležitý fakt, který zneužívají útočníci při *stack overflow*, je ten, že na zásobníku jsou uloženy parametry, které daná funkce dostala, lokální proměnné, a teprve za nimi (na vyšší adrese) je uložena adresa volající funkce – aby bylo jednoduše řečeno, „kam se vrátit“, až bude běh této funkce ukončen. Pokud programátor špatně ohlídá meze některé z proměnných na zásobníku, může útočník přepsat tuto návratovou hodnotu, čímž způsobí, že po opuštění funkce se začne provádět jiný kód, než který by se prováděl v běžné situaci (tzn. začala by se provádět instrukce následující po volání funkce ve volající funkci).

Nejjednodušší „Hello world!“ program, který je zranitelný na *stack overflow*, vypadá takto:

```
int main(int argc, char * argv [])
{
char buf [512];

if (argc>1)
strcpy(buf, argv[1]);
return(0);
}
```

Tomu, kdo útočí na podobný program, stačí jediné – uvědomit si přesně, jak v tomto konkrétním případě vypadá situace na zásobníku, přepsat uloženou adresu, kam se má po skončení funkce vrátit řízení na adresu proměnné `buf`, a do proměnné `buf` vložit (v tomto případě triviálně jako parametr na příkazové řádce) přímo kódované instrukce procesoru. Po opuštění funkce `main()` se provede skok na začátek proměnné `buf`, a procesor začne provádět v ní uložené instrukce. Obsahu proměnné `buf` se říká *shellcode*, protože má typicky za úkol spustit shell (který bude mít samozřejmě práva původní aplikace).

Pokud najde útočník v programu *heap overflow*, je v poněkud obtížnější situaci, protože nemůže přepsat adresu, kam se má předat řízení po skončení programu – musí postupovat jinak – má přístup k proměnným na haldě – musí tedy například přepsat globální ukazatel na funkci, a čekat, až se tento ukazatel použije k volání funkce.

Jak se vyhnout takovýmto chybám ve svých programech?

Používat n-funkce. Podle standardní konvence, standardní funkce, jako je například `strcpy()`, provádějí svoji činnost dokud nenarazí na nulu (konec řetězce). Funkce, kterým je možno parametrem určit maximální počet kopírovaných bajtů, tento problém řeší (`strncpy()`, `strncat()`, atd.). Je ovšem potřeba si dát pozor na postraní efekty (při delším řetězci není zkopírovaná koncová nula, apod.). Ovšem i v případě, že používáme funkce, které omezují počet zkopírovaných znaků, není to samo o sobě zárukou bezpečného kódu. Například v BIND DNS serveru byla svého času tato chyba

```
struct hosten *hp;
unsigned long address;

memcpy(&address, hp->h_addr_list[0], hp->h_length);
```

Sice je zde určeno kolik bajtů se má kopírovat, ale bohužel útočník může měnit hodnotu předávanou jako třetí parametr, určující délku. Přemýšlet při používání indexů.

```
char table[20];

for (i=0; i<=20; i++)
table[i] = ...
```

Je chybné – zapíše se jeden znak za hranici table. A jak bylo ukázáno v Phrack 55, i takováto jednobajtová chyba stačí k úspěšnému útoku.

Pro vstup nepoužívat nebezpečné funkce. Nebezpečnou funkcí je například funkce `gets()` – ta by neměla být nikde používána, protože jí není možné specifikovat maximální možnou délku. Naprosto stejně nebezpečná konstrukce je často používaná

```
scanf("%s", string);
```

Ale u funkcí typu `scanf()` existuje možnost, jak řídit velikost vstupních dat formátovacím řetězcem:

```
scanf("%50s", string);
```

4 Format string

I v tomto případě, stejně jako v předchozím, jde o velmi nebezpečnou chybu, která umožňuje útočnickovi opět přímo měnit obsah paměti. Narozdíl od buffer overflow, kde jsou chyby často způsobeny tím, že programátor „nepřemýšlí“, v tomto případě vstupuje na scénu také lenost. Ke katastrofě stačí, aby programátor místo

```
printf("%s", retezec);
```

napsal

```
printf(retezec);
```

Tyto dvě instrukce mají samozřejmě pro „obyčejné“ řetězcové proměnné naprosto stejný efekt – když není v řetězci v prvním parametru nalezena žádná „formátovací“ instrukce, je obsah řetězce vypsán na obrazovku. Ovšem ještě před vypsáním řetězce funkce `printf()` parsuje první parametr, a hledá v něm formátovací direktivy. Pokud je najde, vyzvedne odpovídající parametr ze zásobníku (kde by jí byl ve standardní situaci předán). Většina programátorů zná a používá pouze základní formátovací direktivy pro výstup (s pro stringy, c pro znaky, d pro čísla, atd). Ovšem formátovací řetězce jsou mnohem bohatší, než se píše v běžných učebnicích programování – například parametr `n`. Manuálová stránka nám o něm poví toto:

```
The number of characters written so far is stored into the
integer indicated by the int * (or variant) pointer argument.
No argument is converted.
```

Tedy pokud `printf()` nalezne při parsování svého prvního parametru parametr `n`, uloží počet již vypsáných znaků tam, kam ukazuje odpovídající ukazatel (parametr funkce `printf`). Tedy je pomocí tohoto parametru možné zapsat do paměti na určené místo určenou hodnotu! Je také v tuto chvíli vhodné připomenout, že toto chování není specifické pouze pro funkci `printf()`, ale například funkce `syslog()`, používaná v Linuxu pro požádání systémového logovacího démona o zápis do log souboru, má naprosto stejnou sémantiku. Jak tohoto využít dále při útoku? Uvažujme následující program:

```
int main(int argc, char **argv){
char buffer[128];
char tmp[] = "\x01\x02\x03";

snprintf(buffer,sizeof(buffer), argv[1]);
buffer[sizeof(buffer) - 1] = 0;
printf("Obsah promenne buffer: [%s]\n",buffer);
}
```

Tento program na první pohled může skutečně vypadat naprosto v pořádku, a skutečně na světě existuje až překvapivé množství programů, které podobné konstrukce používají. Ukažme si teď, jak se bude program chovat v některých situacích (předpokládejme, že je zkompilován s všeříkajícím názvem `a.out` :).

```
$ ./a.out 123
Obsah promenne buffer: Ahoj
$ ./a.out "123 %x"
Obsah promenne buffer: [Ahoj 30201]
$ ./a.out "123 %x %x"
Obsah promenne buffer: [Ahoj 30201 20333231]
```

Co se tu stalo? Funkce `snprintf()` byla formátovacím parametrem `x` požádána, aby odpovídající parametr převedla do hexadecimálního formátu a vypsala.

Jelikož ale žádné parametry nebyly funkci předány, vzala funkce ze zásobníku data, která tam byla již předtím. Ve druhém příkladě jsme dostali 30201 (0x00030201), což je na x86 procesoru (little endian) obsah proměnné tmp (01020300), která bydlí, coby lokální proměnná, na zásobníku. Pomocí dalšího parametru x jsme dostali první 4 bajty proměnné buffer, atd. Tedy za každé další x se posuneme na zásobníku o 4 bajty, a jsme schopni přečíst co na tomto místě paměti leží. Jak bylo ukázáno dříve, je možné pomocí parametru n zapsat na určenou adresu určenou hodnotu. Což v kombinaci s právě ukázaným postupem umožňuje například zjistit návratovou hodnotu na zásobníku (adresu funkce, která naší funkci zavolala), a na tuto adresu pak napsat kód, který chceme spustit (přepsat v paměti instrukce volající funkce). Další možností je například přepsání ukazatele na funkci, který leží na zásobníku, atd. Snad jsou tyto ilustrativní příklady zneužití dostatečnou výstrahou. Někdy nastávají komplikace při zneužívání těchto chyb (například je v některých případech nutné, aby pro použití parametru n byl řetězec předaný printf skutečně velmi dlouhý, aby útočník trefil adresu o kterou má zájem, atd. Tyto problémy jsou také řešitelné, ale to je již mimo rámec této přednášky).

Jak se takovýmto chybám vyvarovat? Jak bylo ukázáno, tyto chyby jsou způsobeny tím, že je uživateli dovoleno libovolně manipulovat s obsahem první proměnné funkcí printf() a podobných. Tedy řešením je vždy striktně používat nezkrácenou variantu s formátovacím parametrem s. Pokud to z nějakého důvodu není možné, je nutné si řetězec kontrolovat proti nebezpečným znakům ve vlastní režii.

5 Race conditions

Klasická definice race condition je, že race condition mezi procesy nastává tehdy, když výsledek operace závisí na tom, jak jsou proloženy instrukce jednotlivých procesů (jak jsou operačním systémem tyto úlohy naplánovány). Typický příklad, takřikajíc ze života: proces chce přistupovat k nějakému systémovému prostředku (souboru, kusu paměti,...) exkluzivně. Zkontroluje, zda-li žádný jiný proces tento prostředek nepoužívá, a pak ho začne používat. K race condition dojde ve chvíli, kdy jiný proces se zájmem o tento prostředek, provede přesně stejný test právě ve chvíli, kdy první proces již provedl test, ale ještě nezačal prostředek využívat. Chyby zmiňované v předešlých kapitolách umožňovaly útočníkovi spustit „svůj“ kód, který si dříve připravil, a který například spustil shell s právy, která vlastnil zranitelný program. Toto není typický případ programů zranitelných pomocí race conditions.

Uvažujme program, který běží s právy uživatele root, a občas potřebuje uložit nějaká data do souboru vlastněného uživatelem, který program spustil. Chybný kód může vypadat například takto:

```
struct stat st;
FILE *f;

if (stat (filename, &st) < 0) {
```

```

    fprintf (stderr, "Soubor %s neexistuje!\n", argv [1]);
    exit(-1);
}
if (st.st_uid != getuid()) {
    fprintf (stderr, "Vlastnik souboru %s neodpovida Tvemu UID!",
            filename);
    exit(-1);
}
if (!S_ISREG (st.st_mode)) {
    fprintf (stderr, "%s Neni jen tak nejaky soubor (symlink?)\n",
            filename);
    exit(-1);
}

if ((f = fopen (filename, O_RDWR)) == NULL) {
    fprintf (stderr, "fopen() selhal!\n");
    exit(-1);
}

    fprintf (f, "%s\n", message);
    fclose (fp);
    fprintf (stderr, "Write OK\n");
    exit(1);

```

Pokud se ve chvíli mezi voláním `stat()` (které zjistí o souboru dostupné informace a naplní jimi strukturu `st`) a podmínkou, která zkoumá, je-li daný soubor symbolický link, podaří útočnickovi soubor zrušit a místo něj vytvořit symbolický link na soubor, který chce přepsat a na jehož přepsání by v běžné situaci neměl práva, ale běžící program je má (například `/etc/passwd`, `.rhosts` u uživatele v adresáři, apod.), je do souboru, na který symbolický link ukazuje, zapsán obsah proměnné `message`.

Problém s tímto typem útoku spočívá v tom, že race condition typicky nastává po poměrně krátkou dobu, a útočník musí mít tedy štěstí, aby takřkajíc „trefil“ tu správnou chvíli, (například pro vytvoření symbolického linku). Typicky se útočník v takovémto případě spoléhá na hrubou sílu, a vytvoří si krátký skript, který stále dokola zkouší spustit program a pak vytvořit symbolický link, přičemž si může vypomáhat různými dalšími akcemi, které mu zajistí zpomalení procesu, na který útočí (aby se race condition vyskytovala po co nejdelší dobu). Například tak, že mu pomocí systémového příkazu `nice` sníží prioritu, případně hodně zatíží systém nějakou nekonečnou smyčkou (`while(1)`), apod.

Abychom z výše uvedeného programu odstranili race condition, upravíme ho následujícím způsobem:

```

struct stat st;
int fd;
FILE * fp;

if ((fd = open (argv [1], O_WRONLY, 0)) < 0) {
    fprintf (stderr, "Soubor %s nelze otevrit\n", filename);
    exit(EXIT_FAILURE);
}

```

```

}
fstat (fd, & st);
if (st.st_uid != getuid()) {
    fprintf (stderr, "Vlastnik souboru %s neodpovida Tvemu UID!",
            filename);
    exit(-1);
}
if (!S_ISREG (st.st_mode)) {
    fprintf (stderr, "%s Neni jen tak nejaky soubor (symlink?)\n",
            filename);
    exit(-1);
}
if ((fp = fdopen(fd, "w")) == NULL) {
    printf("Soubor %s nelze otevrit\n", filename);
    exit(-1);
}
fprintf (fp, "%s", argv [2]);
fclose (fp);
fprintf (stderr, "Write OK!\n");
exit(1);

```

Tento upravený program využívá toho, že jakmile je jednou soubor pomocí volání `open()` otevřený a je mu přiřazen jádrem deskriptor, pak jakékoliv změny (jména, práv), která jsou provedena na tomto souboru, neovlivní již otevřený deskriptor – při otevření dojde k jakémusi svázání obsahu se jménem, takže i když někdo tento otevřený soubor zruší (jeho jméno přestane ve filesystému existovat), program který ho má otevřený s ním může běžným způsobem pracovat. Z tohoto příkladu je dobré si vzít ponaučení, že je vždy vhodné používat volání, která používají deskriptory otevřených souborů, než pouze cesty k souborům. Tedy používat `fchdir()`, `fchmod()`, `fchown()`, `fstat()`, `ftruncate()`,... místo jejich ekvivalentů, které místo deskriptoru mají jako parametr cestu k (neotevřenému) souboru.

Programy často potřebují ukládat dočasná data do souborů na disku. Otvírání dočasného souboru, pokud není uděláno dobře, velice často zavání race condition – i v takových projektech, jakými jsou Apache, `wu-ftpd`, `inn`, `getty`, byly již v historii nalezeny race conditions při práci s dočasnými soubory. Dočasné soubory jsou typicky vytvářeny v adresáři `/tmp`, z několika důvodů – aby systémový administrátor věděl, kde jsou tyto soubory centralizované, a mohl je periodicky promazávat, případně jsou tyto umístěny na speciální parittion (například ramdisku), atd. Adresář `/tmp` se od ostatních běžných adresářů liší svými právy – má nastavený sticky bit, který zajišťuje, že pouze vlastník adresáře, a vlastník souboru v tomto adresáři, mají povoleno soubor smazat, přestože právo na zápis mají do tohoto adresáře typicky všichni. Problém s race conditions při používání dočasných souborů, je předvídatelnost jména souboru, který aplikace použije, čímž usnadňuje útočníkovi provedení útoku přes symbolické linky. Existují knihovní funkce, které poskytují nepředikovatelná jména pro dočasné soubory. Nicméně ani s náhodně generovanými jmény souborů není vhodné používat konstrukce typu

```

if((fd = open(filename, O_RDONLY)) != -1) {
/* soubor se podarilo otevrit */
fprintf(stderr, "Chyba: soubor %s jiz existuje\n",
filename);
exit(-1);
}
fd = open(filename, O_RDWR | O_CREAT, 0644);

```

protože samozřejmě obsahují na první pohled snadno rozpoznatelnou race condition, a pokud by se útočníkovi jakýmkoliv způsobem podařilo naše jméno souboru uhádnout, je pro něj již triviální tuto chybu zneužít. Řešením je používat flag `O_EXCL` spolu s `O_CREAT`, což má za důsledek, že `open()` vrátí chybu, pokud soubor již existuje, ale test a otevření provede atomicky (bez race condition – není možné do tuto akci proložit akci jinou).

6 CGI skripty

Nejtypičtějším útokem proti špatně napsané aplikaci, která s uživatelem interaguje skrz web, je špatná validace vstupních dat, která uživatel může aplikaci poskytnout, a jejíž následná špatná interpretace. Pouze pro inspiraci zde uvedu několik málo příkladů typických nebezpečných zdrojových kódů, kterých je skutečně plný web.

6.1 Cross-site scripting z rychlíku

Hello-world! skript, který je náchylný na typ útoku zvaný *cross-site-scripting* (někdy též označovaný *XSS*), může vypadat například takto:

```

<html>
<body>
<?
  echo Hello $name;
?>

```

Tím, že uživatel může libovolně ovlivnit zvenku obsah proměnné `name`, umožňuje mu to například do stránky vložit javascript. Pokud tato stránka bude používána například pro přihlašování do nějakého systému, může javascript hlídat heslo a po jeho zadání ho někomu oznámit. Ptáte se, proč by si někdo sám vkládal do stránky vhodnou formulací URL takovýto skript? Odpověď je, že k tomu může být oběť donucena – například tak, že jí útočník ze svých stránek (případně HTML mailem) odkáže na příslušnou stránku, ovšem za URL vloží ještě kód pro vložení skriptu, a pak už se může jen radovat, až mu bude oznámeno heslo oběti. Tento útok se i dnes často vyskytuje (donedávna například existoval na titulní přihlašovací stránce služby Yahoo).

6.2 PHP include závislý na proměnné

Doby, kdy tuto chybu obsahoval skoro každý druhý PHP skript jsou již našťěstí minulostí, nicméně i dnes lze (i například za pomoci chytře formulovaných dotazů v Googlu) najít neuvěřitelně velké množství stránek, které touto chybou dosud trpí. O co jde?

```
<html>
<body>
<?
    include($user);
?>
...
</body>
</html>
```

Tento mohl chtít například vložit soubor, který má stejné jméno jako uživatel (například si každý uživatel upraví, jak ho má skript po přihlášení pozdravit – to například umožňují některé webmaily). Problém je, že proměnnou user může uživatel pomocí URL ovládat zcela libovolně. Tedy si například může pomocí

```
http://server.cz/skript.php?user=../../../../etc/passwd
```

nechat do stránky vložit obsah příslušného souboru. A co hůř, funkce include akceptuje i URL... tedy s trochou fantazie si můžeme někde na své IP adrese rozběhnout webserver, který nebude interpretovat PHP skripty, umístit si na něj soubor s obsahem

```
<?
system("....");
?>
```

PHP pak provede include, a interpretuje PHP skript – tedy na systému spustí takový příkaz, jaký se nám, coby útočníkovi, zlíbí.

6.3 (nejen) Perl

Předpokládejme, že chceme napsat skript, který bude přes web ukazovat výstup programu finger pro zadaného uživatele. Naviní skript může vypadat takto:

```
print "<BODY>";
$login = $input{'login'};
$login =~ s/([;<>*\|'&!\#\(\)\[\]\{\}\:'])/\\$1/g;
print "Login $login<BR>\n";
print "Finger<BR>\n";
$CMD= "/usr/bin/finger $login";
open(FILE,"$CMD") || goto form;
print <FILE>
```

Tento skript sice bere ohled na některé obecně známé chybné znaky, které uživatel za žádných okolností nesmí předat aby se dostaly až k shellu (například zpětné apostrofy by shell interpretoval – tedy by spustil příkaz mezi nimi uzavřený, atd, ale zapomíná na line feed (konec řádky) (kromě jiného). Tedy volání skriptu

```
http://server.cz/finger.cgi?login=kmaster%0Acat%20/etc/passwd
```

nám opět prozradí obsah souboru /etc/passwd. Uzavírání parametru do uvozovek také nestačí, protože samozřejmě tyto uvozovky můžeme „zevnitř“ proměnné ukončit.

Podobných útoků existuje celá řada – například pokud se proměnná, jejíž hodnotu uživatel může zvnějšku ovlivnit, používá na webserveru pro formulaci SQL dotazu, a uživatel pomocí středníku v proměnné SQL dotaz ukončí, tak za středníkem může zformulovat svůj vlastní dotaz, a oba se provedou. Podobných triků lze vymyslet nepočítaně, jejich společnou vlastností je nedostatečná validace uživatelem zadaných dat a další používání těchto dat. Správný přístup k řešení tohoto problému nespočívá v určení „nebezpečných“ znaků a ty ze zadaného řetězce eliminovat, protože hrozí, že nějaký nebezpečný bude programátorem opomenut, nýbrž v určení bezpečných znaků, a ty jediné povolit jako možné hodnoty proměnné. Tedy například v perlu:

```
my $safe = '\w\d';
my $danger = '&'\''\\|\"*?~<>^(){}\$%&n\r\[\]' ;

if ($input =~ m/^[${safe}${danger}]+$/g) {
    $input =~ s/([${danger}]+)/\\$1/g;
} else {
    die "Bad input chars\n";
}
print "input = [$input]\n";
```

Znaky, které se vyskytují ve vstupním řetězci a v proměnné safe jsou považovány za bezpečné, a nic se s nimi nedělá. Znaky, které se vyskytují v proměnné danger jsou považovány za potenciálně nebezpečné, a proto je před ně umístěno lomeno. Vstupní řetězce, které obsahují jiný znak než který je obsažen v jedné z těchto dvou množin, je okamžitě odmítnut.

7 Podpora od systému

Není v niších silách přečíst zdrojové kódy všech programů, které běží na jeho počítači, a opravit v nich všechny bezpečnostní chyby. Nicméně existují patche do kernelu, které útočnickům velice znesnadňují práci. Jako jeden příklad za všechny uvedu grsecurity patch, který způsobuje, že stránky paměti, ve který sídlí stack a heap, jsou označeny jako non-executable, a tudíž útočník nemůže použít výše popsané triky (nemůže si umístit do paměti svůj shellcode a pak na něj skočit,

protože procesor odmítne instrukce z těchto oblastí provádět). Dále tento patch umožňuje nastavit taková práva na adresář /tmp, aby bylo nemožné sledovat odtamtud symlinky na soubory nepatřící uživateli, který o to žádá, apod. Tento patch obsahuje nepřehledné množství dalších bezpečnost chránících vlastností (i například co se týče sítě). Viz <http://www.grsecurity.net/>. Dalšími patchi do kernelu, které se zabývají problematikou bezpečnosti a ochranou před špatně napsanými programy jsou <http://www.lids.org/>, <http://www.rsbac.org/>, <http://medusa.terminus.sk/> a další. Každý z těchto patchů má odlišnou filosofii a architekturu, před aplikací je dobré prostudovat ostatní a zvážit, který se hodí pro daný systém nejlépe.