Lokální útoky na operační systém linuxového typu

4.12. | Tomáš Pelka

Článek se zabývá otázkou lokální bezpečnosti, tedy takovou, která je realizována lokálně, nikoliv prostřednictvím sítě. Operační systém Linux byl vybrán z důvodu jeho rostoucí obliby, ale především jeho celkové vyšší transparentnosti a otevřenosti.

Úvod

Bezpečnost je velmi komplexní záležitost. Zjednodušeně by se dala představit jako skládanka, u níž je bezpodmínečně nutné zajistit správné složení do výsledného celku, ve kterém nebudou "díry".

Linux má už ve své podstatě v tomto ohledu dvě protichůdné vlastnosti. Ta kladná vychází z toho, že do něj přispívá velké množství lidí, a tudíž se velmi rychle rozvíjí. Se zmíněnou vlastností koresponduje velmi rychlá oprava chyb. Velké množství přispívatelů ale vytváří stinnou stranu problematiky. Chyby mohou být opraveny, ale také nevědomky do operačního systému zaneseny.

Vyvstává potřeba organizace, která by celý proces kontrolovala. Dojde-li v operačním systému k implementační chybě, je velmi složité ji později napravit. I když původní Unix nebyl přímo navržen s ohledem na bezpečnost, lze dnes unixové systémy, včetně Linuxu, zabezpečit poměrně dobře. Díky otevřenosti zdrojových kódů jádra a programů lze poměrně snadno prozkoumávat výhody a slabiny programů na té nejnižší úrovni a díky licenci GNU GPL či jakékoliv jiné OSS licenci tyto slabiny opravit nebo možnost jejich výskytu omezit. Otevřenost kódu zaručuje, že před námi nikdo nemůže nic skrývat. Navíc otevřenost nám umožňuje Linux vylepšovat a upravovat [1].

Vše však má svoji odvrácenou stranu. Nevýhoda filozofie tkví v tom, že stejně jako uživatelé mají přístup k otevřeným zdrojovým kódům také potenciální útočníci, kteří mohou tyto kódy zneužít ve svůj prospěch.

Používání bezpečných programů je jen část úkolu. Neméně důležité je služby také správně nakonfigurovat.

Pokusme se hackera identifikovat a lépe popsat. Hacker je člověk či raději programátor, který vnímá programování jako jistou formu uměleckého vyjádření a počítač jako umělecký nástroj. Hackeři jsou poháněni snahou po sebezdokonalení a řešení neobvyklých problémů novými cestami, touhou rozebírat a chápat. Tyto hodnoty poháněné vědomostmi se nazývají hackerskou etikou: uznání logiky jako formy umění, podpora svobodného toku informací překonávající všechny hranice a omezení s prostým cílem dosáhnout lepšího chápání okolního světa. Tato etika vychází ze stejné filozofie jako Pythagorejci ve starém Řecku, kteří měli podobnou etiku a subkulturu navzdory neexistenci počítačů [2].

Jak rozpoznat dobrého hackera, který přináší nový, kreativní pohled na technické vymoženosti, od toho špatného, zloděje čísel kreditních karet? Pro špatného hackera byl vymyšlen termín cracker, který popisuje špatného hackera a odlišuje

jej od toho dobrého. Bohužel tento výraz není obecně často užíván. A tedy pojmy hacker a cracker v uších obecné veřejnosti splývají [2].

Exploitování programu

Exploitování je základem hackingu. Program není nic jiného než komplexní sada pravidel sledující určitý tok činností, která počítači říkají, co má dělat. Exploitování programu znamená přinutit programy vykonávat to, co chce útočník, dokonce i když je program navržen tak, aby tomu zabránil. Program může dělat jen to, k čemu je navržen. Bohužel to, co je napsáno se nemusí shodovat s tím, co programátor zamýšlel. Bezpečnostní díry jsou ve skutečnosti vady nebo přehlédnutí v návrhu programu nebo v prostředí, ve kterém program pracuje [1].

Často opakovaná programátorská chyba se nazývá *off-by-one*. Název napovídá, že jde o chybu, při níž se programátor splete o jedničku. Následující příklad vše vysvětlí. Představme si, že máme postavit plot o délce 20 m. Po dvou metrech bude sloupek. Kolik sloupků budeme potřebovat? Samozřejmě, bez dlouhého přemýšlení je odpověď 10. Ale to je špatně. Ve skutečnosti je jich potřeba 11. Někdy se tato chyba nazývá také *fencepost error*. Takovéto chyby často bývají nepostřehnuty, protože programy nejsou standardně testovány na všechny možné vstupní varianty. Jejich efekt se mnohdy neprojeví, ale když už se projeví, může dojít k lavinovému efektu a krom logiky programu může být narušena i bezpečnostní stránka [1].

Jeden nedávný omyl (cca před třemi lety) se přihodil i v OpenSSH. Byla tam nalezena chyba off-by-one v kódu pro alokaci kanálu. Tato chyba byla po objevení hodně exploitována. Kód obsahoval tuto podmínku `if`:

```
if (id < 0 || id > chanells_alloc)
```

a správně měl kód vypadat takto:

```
if (id < 0 || id >= chanells_alloc)
```

Toto je jen jeden příklad z mnoha. Existují ovšem chyby, jejichž exploitování není tak jednoduché. Mezi takové netriviální, obecné exploitovací techniky patří chyby přetečení paměti (*buffer-overflow*) a chyby ve formátovacím řetězci. S těmito chybami je nakonec možné zcela převzít vládu nad průběhem vykonávání programu. Nejběžnější technikou je propašování části škodlivého kódu a jeho spuštění. Takovým technikám se říká *execution of arbitrary code* (spouštění libovolného kódu). V následujících kapitolách bude tato chyba často skloňována.

Lokální útoky - přetečení paměti

V nadcházející části se budeme věnovat především chybně napsanému kódu. Chybně napsaný kód je velmi nepříjemná záležitost. Způsobuje časté havárie, ztráty dat a zbytečné úniky výpočetního výkonu. Jakkoliv může být takový kód při normální práci nepříjemný, zvykli jsme si na něj a počítáme s ním. Často zálohujeme, stahujeme aktualizace a bezpečnostní záplaty. Skutečný problém ale nastává, když je chyba v programu potenciálně zneužitelná útočníkem. Chybný kód představuje vstupní bránu pro řadu útoků, ať už lokálních nebo vzdálených. Existuje několik tříd programátorských chyb, které mohou způsobit vážné trhliny v bezpečnosti služeb nebo celého systému.

Největší problém nastává u programů, které mají nastaven suid respektive sgit bit. Takové programy by měly být programovány co nejpečlivěji, protože program běží s jinými právy, než s jakými byl původně spuštěn. Uživatel využívající takový program může kvůli chybě získat práva, která by vůbec získat neměl. U tohoto druhu programů je třeba dbát na to, aby byla práva odkládána ve správném čase a hlavně korektně. Nejúčinnější ochranou je dobrá a rychlá informovanost. Jen tak je možné provést odpovídající kroky vedoucí k nápravě.

Paměť

Dříve, než budou vysvětleny konkrétní programátorské chyby, které jsou využitelné k exploitování, je třeba říci pár slov o paměti, její segmentaci a deklaraci.

Paměť je dočasné úložiště, nic víc než bajty označené tzv. adresou. Bajty na jednotlivých adresách lze číst nebo zapisovat.

Procesor má vlastní paměť, která je relativně malá. Tyto části paměti se nazývají registry. Existují speciální registry, které udržují informace o vlastním spouštěném programu. Jeden z nejdůležitějších je EIP (extended instruction pointer). Jak název napovídá, jedná se o ukazatel ukazující na právě prováděnou instrukci. Dalšími důležitými registry pro vykonávání programu jsou EBP (extended base pointer) a ESP (extended stack pointer) [2]. Tyto informace jsou platné pouze pro procesory Pentium a kompatibilní.

Důležitou věcí v souvislosti s deklarací paměti je pořadí bajtů v čtyřbajtovém slovu na architektuře x86. Toto řazení je známé jako little endian, které říká, že nejméně významný bajt je první. Znamená to, že bajty slova, jsou uloženy v převráceném pořadí. Hexadecimální hodnota `0x12345678` v kódování little endian je v paměti uložena jako `0x78 0x56 0x34 0x12`.

Občas se stane, že pole alokovaných bajtů není využito úplně. Za alokovanými bajty budou nevyužité znaky. V tom případě se použije nula (bitová nula nebo také znak `null`) pro ukončení řetězce. Ukončování řetězců nulovou hodnotou zvyšuje efektivitu a umožňuje funkcím s řetězcem lépe pracovat.

Paměť programu je rozdělena na pět segmentů: `text`, `data`, `bss` (block started by symbol, segment neinicializovaných dat), heap (halda) a `stack` (zásobník). Každý segment reprezentuje speciální část paměti, která je vymezena pro určité účely [3].

Segment `text` je někdy označován jako code. Jde o místo, kde se nacházejí instrukce strojového jazyka. Tato vlastnost však platí jen pro procesory vycházející z Von Neumannovy architektury, kde není oddělena paměť na datovou a instrukční. Naproti tomu procesory s Harvardskou architekturou mají zmiňované paměti odděleny, tudíž tento bezpečnostní problém odpadá. Navíc novější procesory obsahují ve své instrukční sadě příznak, který tyto paměti odlišuje.

`Data` a `bss` se využívají k ukládání globálních a statických proměnných.

Segment heap (halda) se používá pro ostatní programové proměnné. Celá tato paměť je řízena alokačními a de-alokačními algoritmy, které rezervují paměťové místo pro další použití.

Segment `stack` (zásobník) má také proměnnou velikost a používá se jako dočasné úložiště pro kontext volání funkcí. Segmenty heap i stack jsou

dynamické a rostou proti sobě.

Přetečení paměti

Programovací jazyk C nepatří mezi jazyky silně typované. Odpovědnost za datovou integritu je ponechána na programátorovi. Kdybychom tuto odpovědnost nechali na překladači, výsledný kód by byl neúměrně velký. Optimalizace a efektivita výsledného kódu je cenou za zvýšenou pečlivost programátora. Kontrola programátora také zvyšuje náchylnost na přetečení paměti. Jakmile je proměnná v alokované paměti, neexistuje žádný mechanismus, který pohlídá, že se proměnná opravdu do alokované paměti vejde. Popsané skutečnosti jsou hlavním rozdílem oproti jazykům opravdu silně typovaným jako například C++ či Java. Máme-li například následující kus kódu:

```
void overflow_function(char *str){
    char buffer[20];
    strcpy(buffer, str);
}
int main() {
    char big_string[128];
    int i;
    for(i = 0; i < 128; i++){
         big_string[i] = 'A';
    }
    overflow_function(big_string);
    exit(0);
}
```

Výsledkem po kompilaci a spuštění je:

```
user@host:~$ gcc overflow.c -o overflow
user@host:~$ ./overflow
Neoprávněný přístup do paměti (SIGSEGV)
```

Ve funkci byl vytvořen buffer o velikosti 20 znaků, to je námi očekávaná hodnota. Do tohoto bufferu však vložíme 128 znaků. Z důvodu přetečení paměti program selže. Tato situace nahrává hackerovi. Programátor nepředpokládal, že někdo bude jako vstupní parametr funkce zadávat tolik znaků. Program, který jinak funguje naprosto normálně, najednou "spadne". Tato skutečnost programátora jen utvrdí v použití vstupního filtru, validátoru dat.

Přetečení zásobníku

Vraťme se ještě k předchozímu programu. Když se zavolá funkce `overflow_function`, vytvoří se na stacku rámec zásobníku. Když se funkce poprvé zavolá, vypadá rámec podobně jako na obrázku 1.1.

Když se funkce pokusí zapsat 128 bajtů dat do 20 bajtového bufferu, dojde k zapsání 20 bajtů do bufferu a zbylých 108 bajtů přepíše návratovou adresu, ukazatel na rámec a argument funkce. Když podprogram skončí, pokusí se pomocí záznamu o návratové adrese skočit zpět do hlavní funkce `main()`, ale tato adresa je přepsána nesmyslnou hodnotou, v našem případě samými `A`, což je hexadecimálně `0x41`. Skočí tedy na adresu `0x41414141`, která je neplatná. V každém případě program skončí. Tento jev je nazýván přetečení zásobníku (stack-based overflow).

K přetečení může dojít i v jiném segmentu paměti. Ale přetečení zásobníku je z

hlediska využití nejzajímavější, protože můžeme přepsat návratovou adresu a tím změnit tok vykonávání programu. Kdyby totiž byla návratová adresa jiná než 0x41414141 a byla by to platná adresa nějakého kódu, který eventuálně podstrčí útočník, znamenalo by to vykonávání právě podstrčeného kódu. Tato metoda se nazývá tzv. infekce kódu (bytecode infection). Bytecode je kus chytře podstrčeného assemblerovského kódu, který je vložen do bufferu. Takový kód má několik omezení. Musí být samostatný a nesmí obsahovat speciální znaky v instrukcích, protože by měl vypadat jako data v bufferu.

Obrázek 1.1: Rámec zásobníku

Jeden z nejpoužívanějších kódů je tzv. shellkód, který spouští příkazový interpret shell. Podaří-li se přelstít nějaký suid root program (takový program, který vlastní root a má přidělen suid bit) tak, aby se s jeho právy spustil shell, má útočník vyhráno a získává plnou kontrolu nad strojem a systémem.

Vycházíme-li ze situace, že máme k dispozici takto postižený program, postačí jen vhodně vygenerovat data, která předáme programu. Data budou obsahovat shellkód, který po skončení programu přepíše návratovou adresu a umožní tak převzít kontrolu nad strojem jako s právy uživatele root. Aby byl shellkód spustitelný, je žádoucí předem znát jeho adresu, což může být u dynamického zásobníku poněkud problematické. Pro správnou funkci je nutné přepsat pouze čtyři bajty návratové adresy. Pokud tato podmínka není splněna, program je ukončen. Pro řešení této situace existují dvě známé techniky.

První je nazvána NOP sled. NOP je instrukce, která nedělá vůbec nic. Použijeme ji ve formě velkého pole NOP, které je ukončeno shellkódem. EIP bude neustále inkrementován, až narazí na shellkód.

Druhou technikou je zaplnění konce bufferu mnoha po sobě jdoucími návratovými adresami. Vytvořený buffer bude vypadat jako na obrázku 1.2.

Obrázek 1.2:Buffer

U obou technik je ale nutné znát alespoň přibližné umístění bufferu v paměti, aby bylo možné uhodnout návratovou adresu. Jednou z možností je aproximování umístění v paměti na základě ukazatele na zásobník (registr ESP). Odečtením

offsetu od ESP získáme relativní adresu libovolné proměnné. Jelikož je prvním prvkem na zásobníku buffer, který je přepsán, správná adresa by měla být ukazatel na zásobník, takže offset by měl být blízko 0.

Přetečení v segmentech halda a bss

Nejen u zásobníku může dojít k přetečení. Stejný problém se týká i haldy a segmentu bss. Tento typ útoků není tak standardizovaný jako přetečení zásobníku, ale to neznamená, že není stejně účinný. Jelikož v tomto případě neexistuje návratová adresa, která by mohla být přepisována, závisí tyto útoky na přepisování některých důležitých proměnných, jež se nacházejí v paměti za bufferem. Pokud je například za bufferem uložen seznam přístupových práv nebo autentizační informace, může přetečení bufferu znamenat získání plných přístupových práv (v nejhorším případě). V jiných případech se může jednat o ukazatel na funkce obsažené v těle programu. Přetečením bufferu a přepsáním těchto adres můžeme spustit shellkód. Protože exploitování v segmentech halda a bss je mnohem více závislé na rozložení paměti v programu, není tak triviální nalézt zranitelná místa programu.

Jeden poměrně chytrý způsob, jak pomocí tohoto druhu přetečení získat superuživatelská práva, ukazují následující řádky. Využijeme jednoduchého programu pro zápis do souboru. Podmínky: nastavení suid bitu, vlastnictví roota a samozřejmě náchylnost k tomuto druhu přetečení. Napaden bude velmi důležitý soubor `/etc/passwd`. Právě pomocí tohoto přetečení bude vytvořen nový uživatelský účet. Prostým editováním `/etc/passwd`. Tento účet bude bez hesla a bude superuživatelský (uid bude rovno 0). Odpovídající řádka v `/etc/passwd` by mohla vypadat následovně.

```
r00t::0:0:hacker:/root:/bin/bash
```

Vkládaný řetězec je nutné drobně upravit. Je důležité, aby data měla správnou délku, která způsobí správné přetečení. Jinými slovy, musí přesně padnout do masky rámce zásobníku.

Lokální útoky - formátovací řetězce

Podobně jako u metod využívajících přetečení paměti je cílem této metody změna toku programu. Exploity tohoto druhu závisejí taktéž na programových chybách a na rozdíl od přetečení paměti nemusejí mít přímý dopad na bezpečnost. Ve srovnání s exploitačními technikami postavenými na přetečení paměti jsou techniky formátování řetězce poměrně snadno odstranitelné [2]. Ovšem za předpokladu, že se o těchto chybách ví.

Formátovací řetězce a funkce printf()

Formátovací řetězce nacházejí uplatnění ve formátovacích funkcích, např. ve funkci `printf()`. Příkladem může být následující kus kódu.

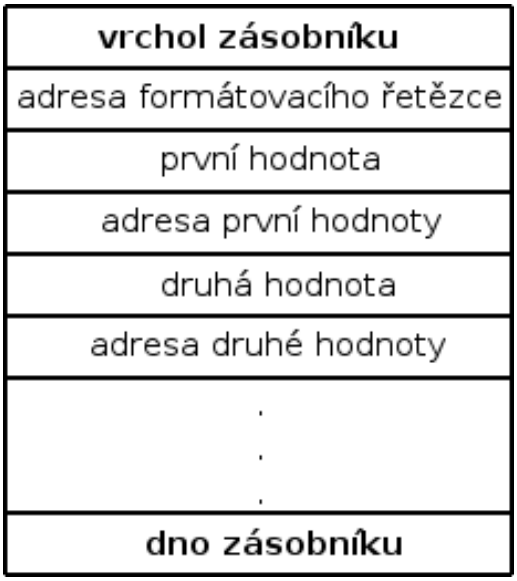
```
printf("Vypiseme neco na standardni vystup:%d",some_var);
```

`"Vypiseme neco na standardni vystup:%d"` je formátovací řetězec. Funkce jej vytiskne, ale nejprve provede speciální operaci, projde jej a otestuje na výskyt formátovacích parametrů. V příkladu je formátovacím parametrem `%d`. Tento parametr zapříčiní vypsání argumentu `some_var`, tedy hodnoty proměnné jako

celé číslo v desítkové soustavě. Většina parametrů získává hodnotu proměnné předáním hodnoty. Existují i výjimky, které očekávají ukazatel. Jedná se především o řetězcové proměnné a data ve formě bajtů. Speciální činnost funkce zmíněná na začátku kapitoly proběhne vždy, když funkce najde ve formátovacím řetězci formátovací parametr. Každý formátovací parametr očekává samostatnou proměnnou. Funkce má proměnný počet parametrů. Důležitá je také skutečnost, že argumenty funkce jsou uloženy na zásobníku v opačném pořadí [4]. Rámec zásobníku je vyobrazen na obrázku 1.3.

V případě, že se budeme zabývat zobrazováním čísel, je jasné, že číslo bude fyzicky uloženo stále stejně ve formátu dvojkového doplňku. To, co ovlivňujeme formátovacím parametrem, je jeho reprezentace.

Vraťme se nyní k počtu parametrů. Bylo řečeno, že je třeba uvést tolik hodnot, kolik je formátovacích parametrů. Co však nastane v případě, že jeden nebo více parametrů chybí? Funkce jednoduše vytiskne hodnotu zásobníku na odpovídající pozici. Právě této vlastnosti je při útocích tohoto typu využito. Zjednodušeně řečeno měníme počet argumentů předávaných nebo očekávaných formátovací funkcí.



Obrázek 1.3: Rámec zásobníku funkce `printf()`

Někdy se stane důsledkem nepozornosti, že programátor namísto `printf("%s",string)` zapíše pouze `printf(string)`. Funkčně je vše v pořádku, formátovací funkce vypíše všechny značky řetězce dané adresou `string`. Funkce končí, když narazí na nulový bajt. Následuje jednoduchý program pro demonstraci výše zmíněné chyby.

```
#include <stdlib.h>
int main(int argc,char *argv[]){
char text[1024];
static int testVal=-72;
if(argc < 2){
printf("Pouziti:%s<text k vytisknuti>\n", argv[0]);
exit(0);
}
strcpy(text, argv[1]);
//spravny způsob vytisteni
printf("Spravne:\n");
printf("%s",text);
printf("\n");
```

```
//nespravny způsob vytisteni
printf("Spatne:\n");
printf(text);
printf("\n");
//debug
printf("[*]testVal@0x%08x = %d0x%08x\n",&testVal,testVal,testVal);
exit(0);
}
```

Oba způsoby fungují korektně, pokud programu předložíme jakýkoliv text bez formátovacích parametrů. Změna ovšem nastane, pokud programu předáme parametr, jenž obsahuje formátovací řetězec.

```
user@host:~$ ./format text\%x
Spravne:
text%x
Spatne:
textbf800828
[*]testVal@0x08049728=-720xffffffb8
```

Program vyhodnotil součást řetězce `text%x` jako další formátovací parametr, avšak bez příslušné proměnné ve formátovací funkci. Program tedy vypsal obsah paměti následující za proměnnou `testVal` na zásobníku a tuto hodnotu formátoval jako hexadecimální číslo. Postup je možné využít opakovaně k prozkoumání obsahu zásobníku. Postačí vhodně zvolit parametr. Například požadujeme-li prozkoumání 50 paměťových míst za poslední validní hodnotou, předáme parametr například ve tvaru `perl -e print "%80x."x50;` [2].

Příklad, který byl uveden, ukazoval vypsání paměti pomocí parametru předávaného hodnotou (%x). Použijeme-li však parametr předávaný odkazem (%s), pokusí se program vypsat obsah paměti na adrese dané parametrem programu. Pokud je zvolena neplatná adresa, program havaruje. Pokud je adresa platná, bude vypsán obsah na této paměťové adrese.

Čtení obsahu rámce zásobníku není příliš zajímavé, daleko vhodnější je možnost zápisu do něj. Je použita naprosto stejná technika jako při čtení zásobníku; jedinou odlišností je použití jiného formátovacího parametru. Jedná se o parametr `%n` [4].

Závěr

Další, velmi výhodnou možností je použití "Warning options" překladače GCC. Jelikož nemám žádné znalosti jiných překladačů, zaměřím se pouze na GCC. Mezi tyto "Warning options" nebo "Warning flags" patří například `-Wall`, `-Wformat`, `-Wno-format-extra-args` nebo `-Wformat-security`. Tyto přepínače mimo jiné kontrolují formát argumentů některých funkcí (`printf()`, `scanf()`, ...) nebo varují při možnosti vzniku bezpečnostních problémů [5].

Popsané lokální útoky nebo lépe řečeno programátorské chyby lze většinou poměrně snadno odstranit. Daleko větším problémem je jejich nalezení. Díky komunitě, která se kolem Linuxu rozrostla, není tento problém neřešitelný. Je kladen poměrně velký důraz na otevřenost i v otázkách bezpečnosti, nic není skrýváno. Tato skutečnost napomáhá tomu, aby byly chyby brzo nalezeny a odstraněny. Doporučením pro předcházení problémům je častá aktualizace systému a sledování diskusních fór, konferencí a jiných informačních zdrojů s tématikou bezpečnosti. Mimo tyto základní metody je vhodné pro další ochranu

před tímto druhem útoků systematicky získávat dobré programátorské návyky, pečlivé prověřování a testování napsaného kódu.

V poslední době je stále oblíbenější řešení knihovna Libsafe. Obrovskou výhodou je jednoduchá instalace. Navíc není nutné měnit již zkompilované programy. Podle autorů zastaví tato knihovna drtivou většinu chyb způsobených přetečením bufferu. Libsafe chrání pouze provádění aplikací, ne samotného jádra operačního systému. Tento kód chrání jedna z alternativ Libsafe systému OpenWall. Jedná se o jadernou záplatu (kernel patch). Řeší jisté bezpečnostní problémy na úrovni jádra, například vytváří tzv. nespustitelný zásobník (nonexecutable user stack area), čímž zabraňuje spouštění kódu v zásobníku. Zaměřuje se především na otázky spojené se správou paměti (nechrání haldu).

Další alternativou k Libsafe je například StackGuard.

Knihovna Libsafe pracuje následovně. Namísto běžně používaných funkcí jazyka C, jako je například `gets()`, `strcpy()`, `getwd()`, `scanf()` nebo `sprintf()`, vyvolá službu Libsafe. Při tomto volání zjistí, zda-li se cíl volání nachází na zásobníku, a také jestli některá z adres nepřepíše data mimo rámec zásobníku. V případě, že jsou předcházející podmínky splněny, je operace zablokována [1].

Zmíněné funkce provádějí přesně to, co programátor požaduje. Tato vlastnost je důvodem jejich nebezpečnosti. Je dosaženo vysoké rychlosti na úkor bezpečnosti. Druhým problémem je skutečnost, že programátor není schopen zadat velikost bufferu. Nabízí se další řešení: používat bezpečné verze funkcí jako `fgets()` místo `gets()` nebo `strncpy()` namísto `strcpy()` [1].

Lokální útoky nejsou samozřejmě tvořeny jen přetečením paměti a chybami formátovacího řetězce. Dále je možné setkat se s útoky zvanými návrat do libc (*Return-to-libc attacks*), přetečení celých čísel (*Integer overfows*), chyby souběhu (*Race conditions*) atd.

Literatura

- [1] TOXEN, Bob. Bezpečnost v Linuxu: Prevence a odvracení napadení systému. Brno: Computer Press, 2003. 876 s. ISBN 80-7226-716-7.
- [2] ERICKSON, J. HACKING - umění exploitace. Brno: Zoner PRESS, 2005. 263 s. ISBN 8086815-21-8.
- [3] HATCH, B. - LEE, J. - KURTZ, G. HACKING bez tajemství - Linux. Brno: Computer Press, 2003. 644 s. ISBN 80-7226-869-4.
- [4] HEROUT, P. Učebnice jazyka C. České Budějovice: Kopp, 2001. 263 s. ISBN 80-85828-21-9.
- [5] Contributors to GCC. Manuálové stránky GCC, [poslední aktualizace 18. 4. 2007].