# Software a jeho bezpečnosť

Michal Rjaško

Úvod do informačnej bezpečnosti

# Kvizová otázka

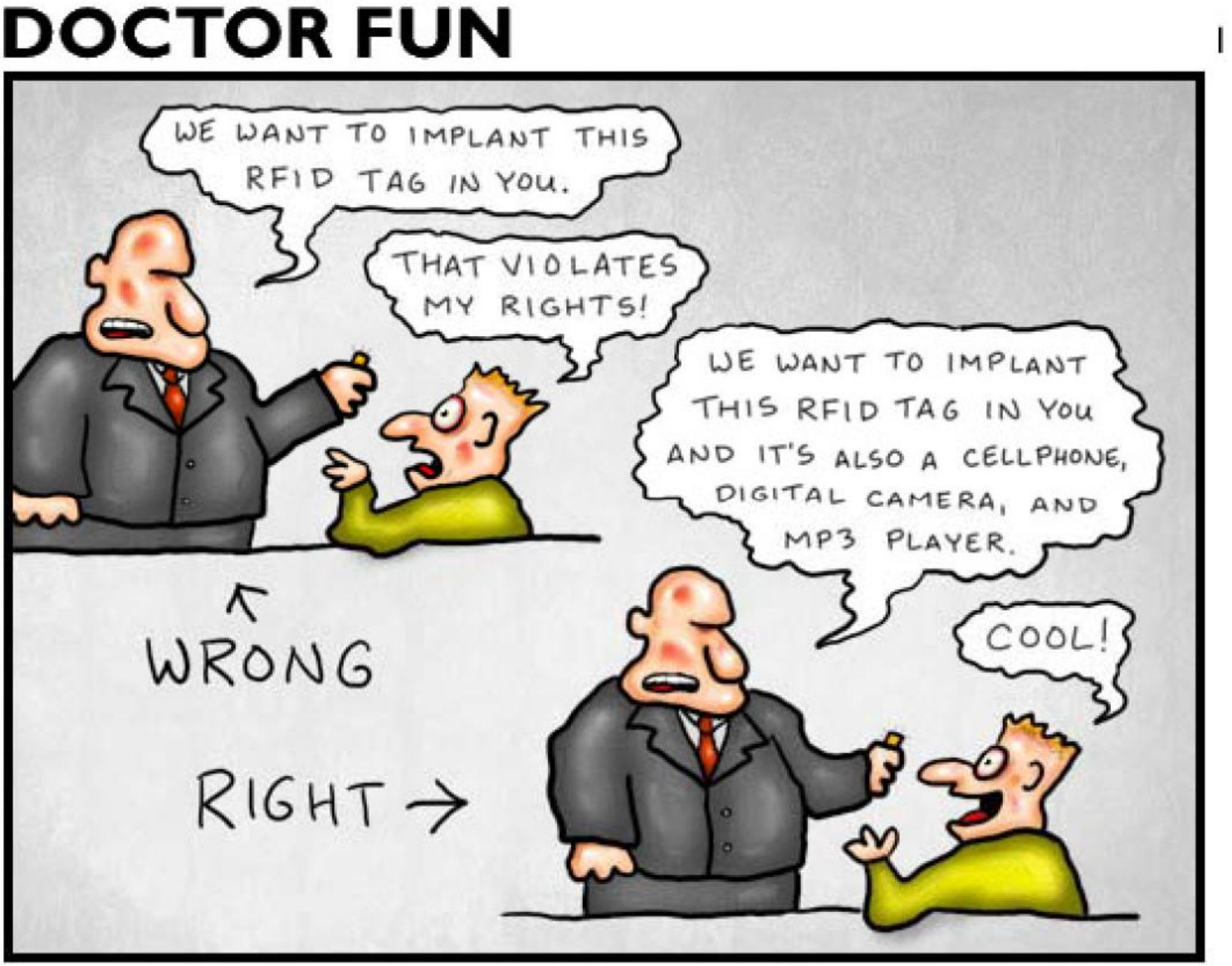
- Koľko na naučilo programovať v C / C++
  - Koľkých je to prvý programovací jazyk?
- Koľko z kurzov, ktoré ste absolvovali Vás
  - upozornilo na chyby typu „buffer overflow“?
  - naučilo vyhýbať sa im?
- Príčinou „nebezpečného“ software sú ľudia
  - Malé povedomie o bezpečnostných  hrozbách
  - Malé znalosti programovacieho jazyka

# Bezpečnosť je vždy **druhoradým** cieľom

- **Primárnym** cieľom software je poskytovať funkcionalitu resp. služby
- Manažovanie vyplývajúcich rizík je odvodený / druhoradý problém
- Funkcionalita je o tom, čo **má** software robiť
- Bezpečnosť je o tom, čo software **nemá** robiť

*Kým nerozmýšľate ako útočník, neuvedomíte si potenciálne riziká*

# Bezpečnosť je vždy druhoradým cieľom



# Funkcionalita vs. bezpečnosť

"After writing PHP forum software for three years now, I've come to the conclusion that it is basically impossible for normal programmers to write secure PHP code. It takes far too much effort. .... PHP's raison d'etre is that it is simple to pick up and make it do something useful. There needs to be a major push ... to make it safe for the likely level of programmers - newbies. Newbies have zero chance of writing secure software unless their language is safe. ... "

[Source http://www.greebo.cnet/?p=320]

Verejný nepriateľ č. 1

# BUFFER OVERFLOWS

# Základ problému

- Predpokladajme, že v C programe máme pole veľkosti 4

```
char buffer[4];
```

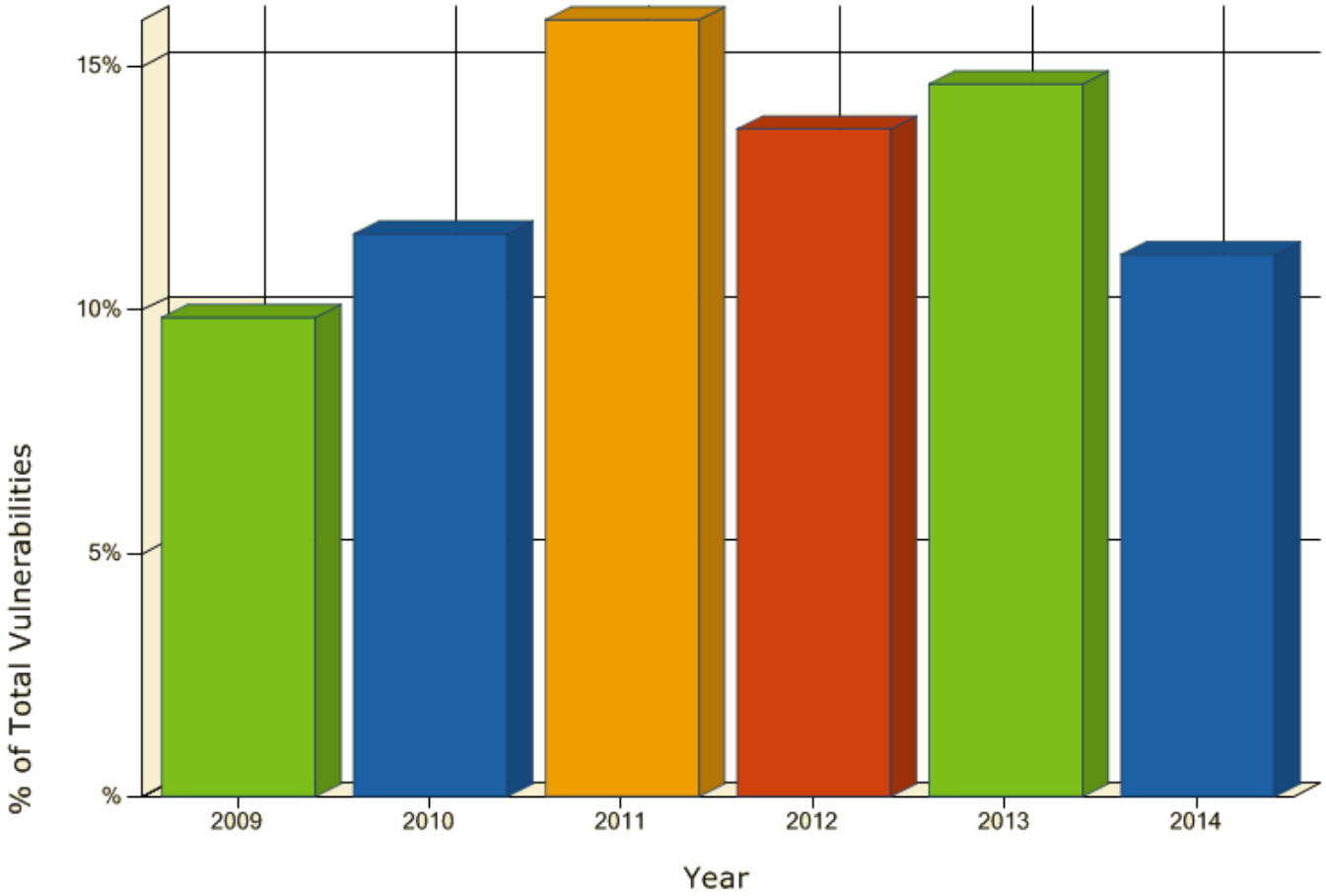
- Čo sa stane po vykonaní nasledovného príkazu?

```
buffer[4] = 'a';
```

- Môže sa stať hocičo
  - Ak ukladané dáta (t.j. 'a') kontroluje útočník, môže si robiť čo chce

# Buffer overflow

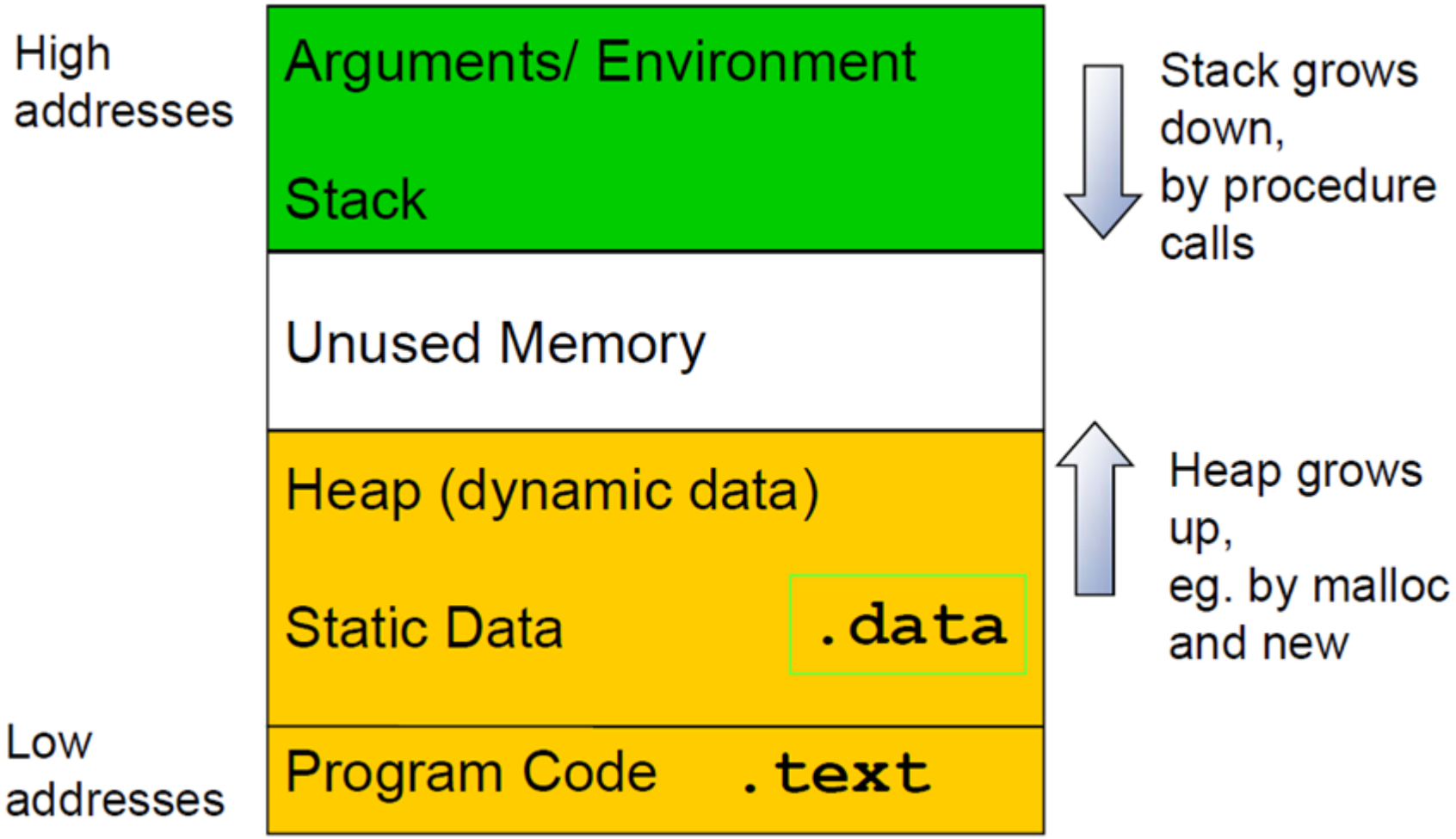
- Buffer overflow patria medzi najčastejšie chyby



# Správa pamäte C/C++

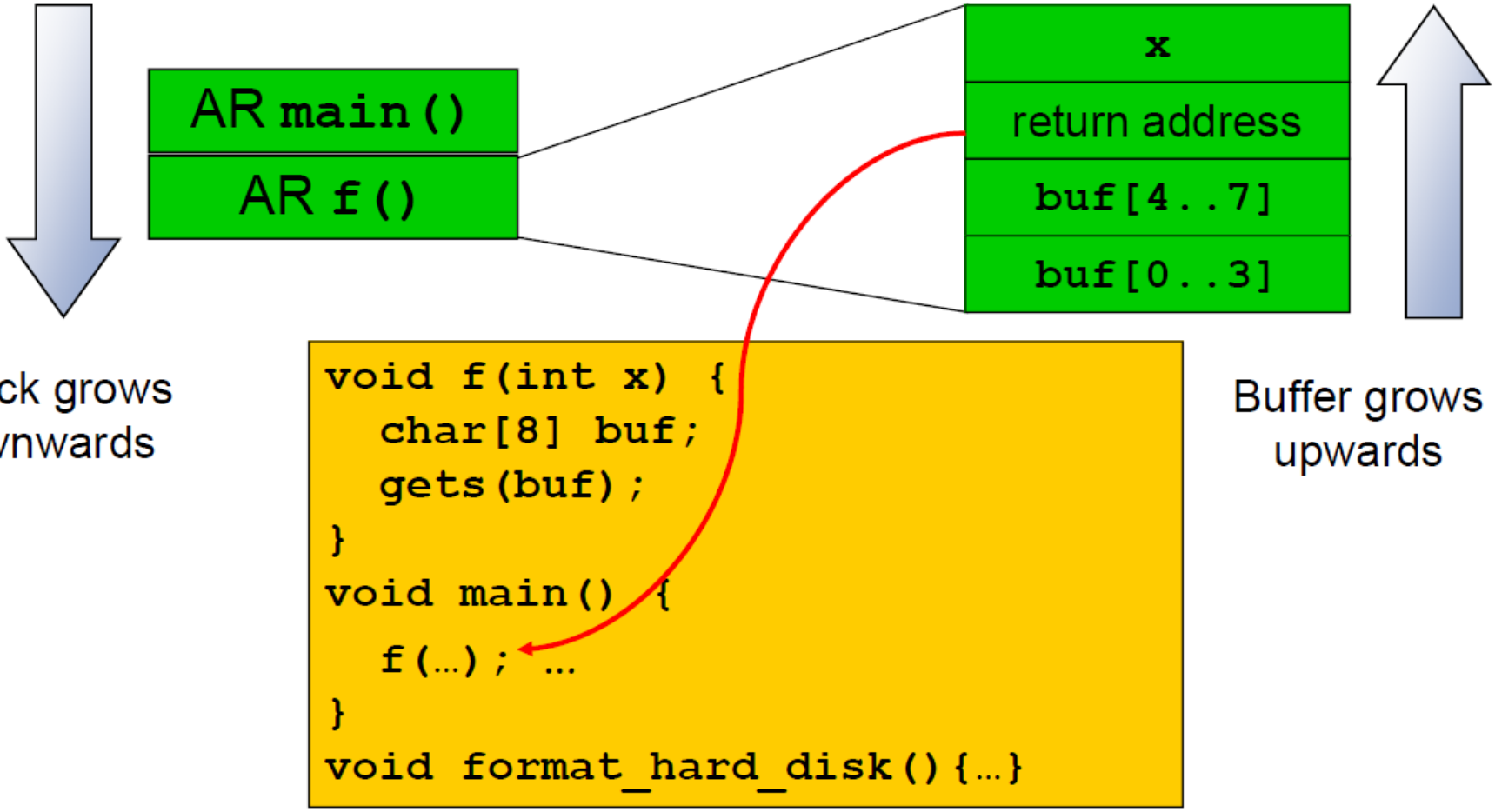
- Program je zodpovedný za správu svojej pamäte
- „Manuálne“ spravovať pamäť je veľmi náchylné na chyby
- C / C++ neposkytujú „**memory-safety**”
- Typické bugy:
  - Zápis mimo rozsahu poľa
  - Problémy so smerníkmi
    - Chýbajúca inicializácia, zlá aritmetika, použitie po dealokácii, zabudnutá dealokácia,..
  - Z dôvodu efektívnosti tieto bugy nie sú detekované počas run-time.

# Rozloženie pamäte procesu



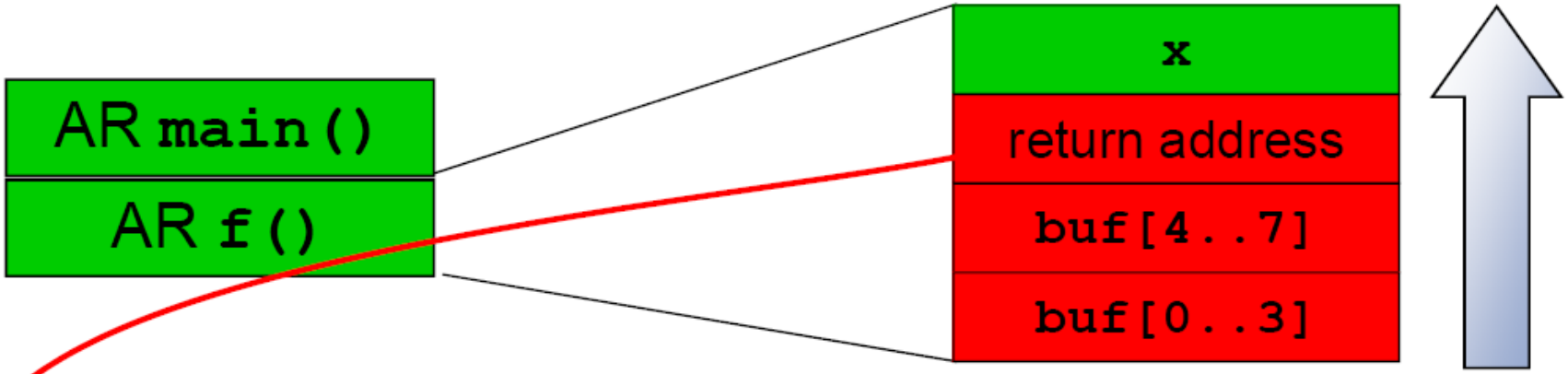
# Stack overflow

- Stack pozostáva z „**Activation Records**“:



# Stack overflow

- Čo ak gets() prečíta viac ako 8 bytov?

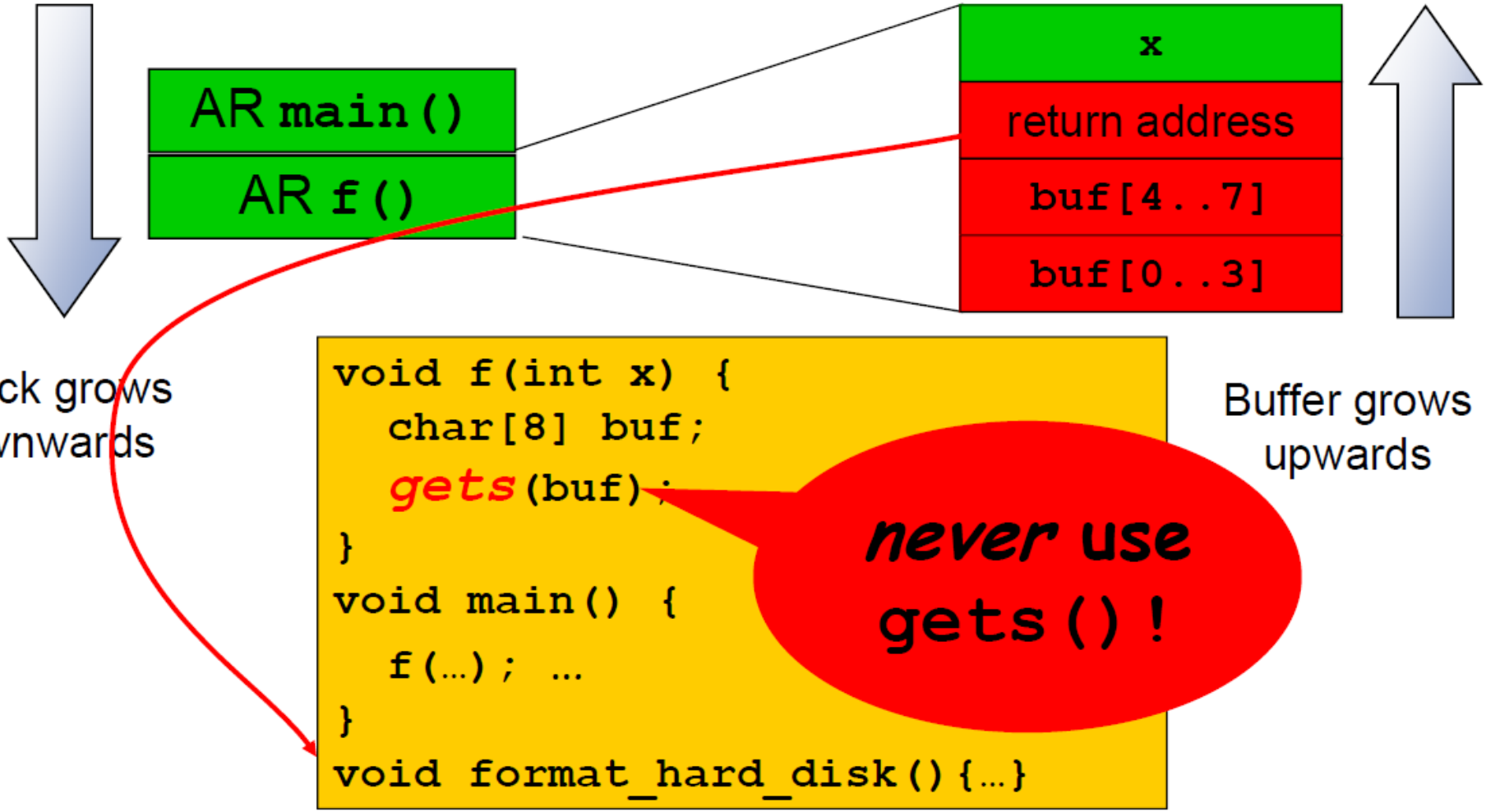



```
void f(int x) {
   char[8] buf;
   gets(buf);
}
void main() {
   f(…); …
}
void format_hard_disk(){…}
```

Buffer grows upwards

# Stack overflow

- Čo ak gets() prečíta viac ako 8 bytov?



# Stack overflow

- Technika útoku: využiť pretečenia buffera na úpravu dát
- Závisí na veľa ďalších detailoch:
  - Napr. ako vyplniť správnu návratovú adresu:
    - Falošná návratová adresa musí byť presne umiestnená
    - Útočník nemusí poznať  ani adresu svojich premenných
  - Prepísané dáta sa nesmú použiť pred návratom z funkcie (mohlo by dôjsť ku pádu programu)
  - …
- Variant: **Heap overflow** využíva heap namiesto zásobníka

# Príklad: fgets

- Nepoužívať gets
- Namiesto toho pužite fgets(buf, size, stdin)

```
char buf[20];
gets(buf); // read user input until
           // first EoL or EoF character
```

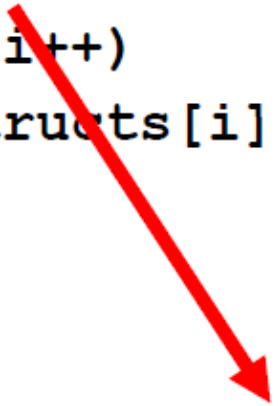
# Príklad: strcpy

```
char dest[20];
strcpy(dest, src); // copies string src to dest
```

- `strcpy` predpokladá, že dest je dostatočne dlhé
- Používať `strncpy(dest, src, size)`

# Príklad

```
bool CopyStructs(InputFile* f, long count)
{    structs = new Structs[count];
     for (long i = 0; i < count; i++)
        { if !(ReadFromFile(f,&structs[i]))
              break;
        }
 }
```



effectively does a
`malloc(count*sizeof(type))`
which may cause integer overflow

- Integer overflow môže spôsobiť buffer overflow

# Príklad

```
#include <stdio.h>

int main(int argc, char* argv[])
{   if (argc > 1)
      printf(argv[1]);
    return 0;
}
```

- Program je zraniteľný cez tzv. format string útok

# Format string útok

- Iný príklad ako poškodiť zásobník
- Reťazce môžu obsahovať špeciálne znaky, ako `%s`
  - `printf("Cannot find file %s", filename);`
- Čo sa stane, ak vykonáme nasledovný kód?
  - `printf("Cannot find file %s");`
- Čo sa stane, ak vykonáme
  - `Printf(string);`
  - Kde string je zo vstupu od používateľa?

# Format string útok

- %x načíta a vypíše 4 byty zo zásobníka
  - môže dôjsť k úniku citlivých dát
- %n zapíše počet výpísaných znakov do zásobníka

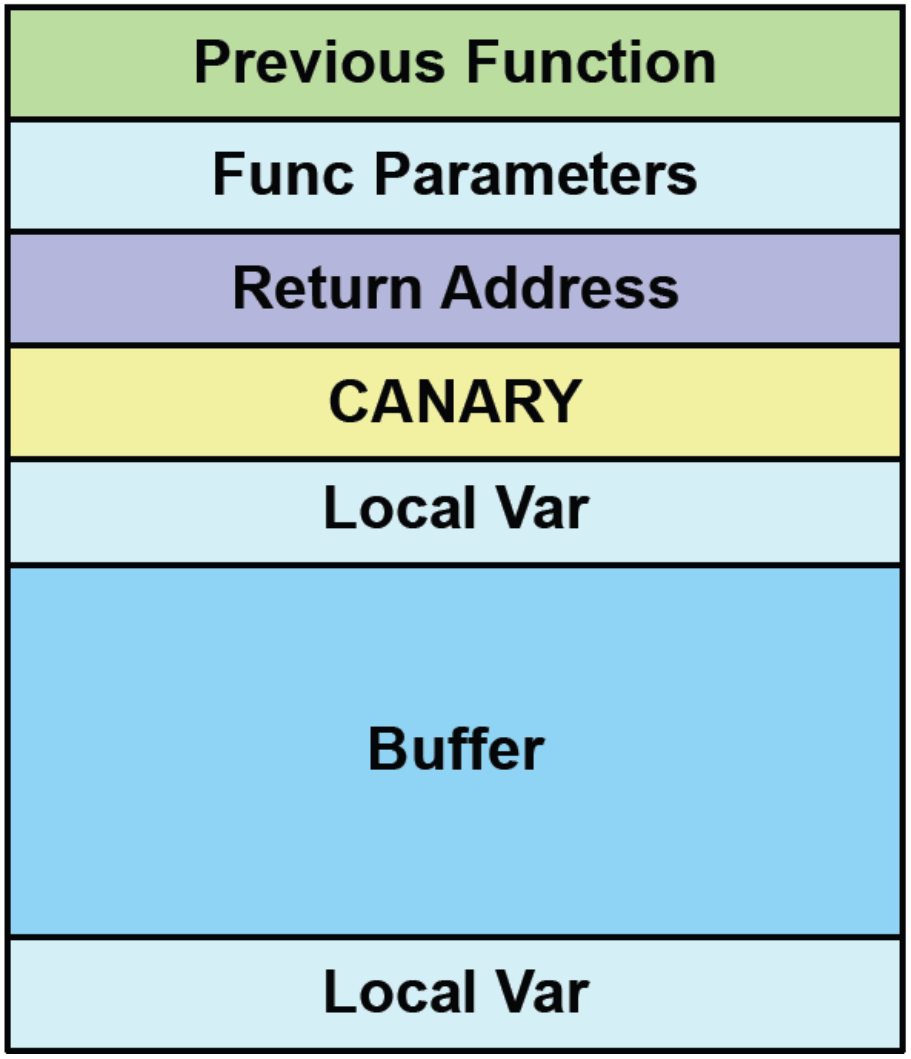
- Format string útok je ľahké ošetriť
  - Namiesto `printf(str)`
  - Použiť `printf("%s", str)`

# RUNTIME / DYNAMICKÁ OBRANA

# „stack canaries“

- „Dummy“  hodnota – kanárik – je zapísaná do zásobníka pred návratovú adresu a skontrolovaná, keď funkcia vracia hodnotu
- Obyčajné pretečenie zásobníka prepíše aj kanárika, čo môže byť detekované
- Obozretný útočník však môže zapísať do kanárika správnu hodnotu.
- Možné vylepšenia:
  - Použiť náhodnú hodnotu pre kanárika
  - XOR náhodnej hodnoty s návratovou adresou

# „stack canaries“

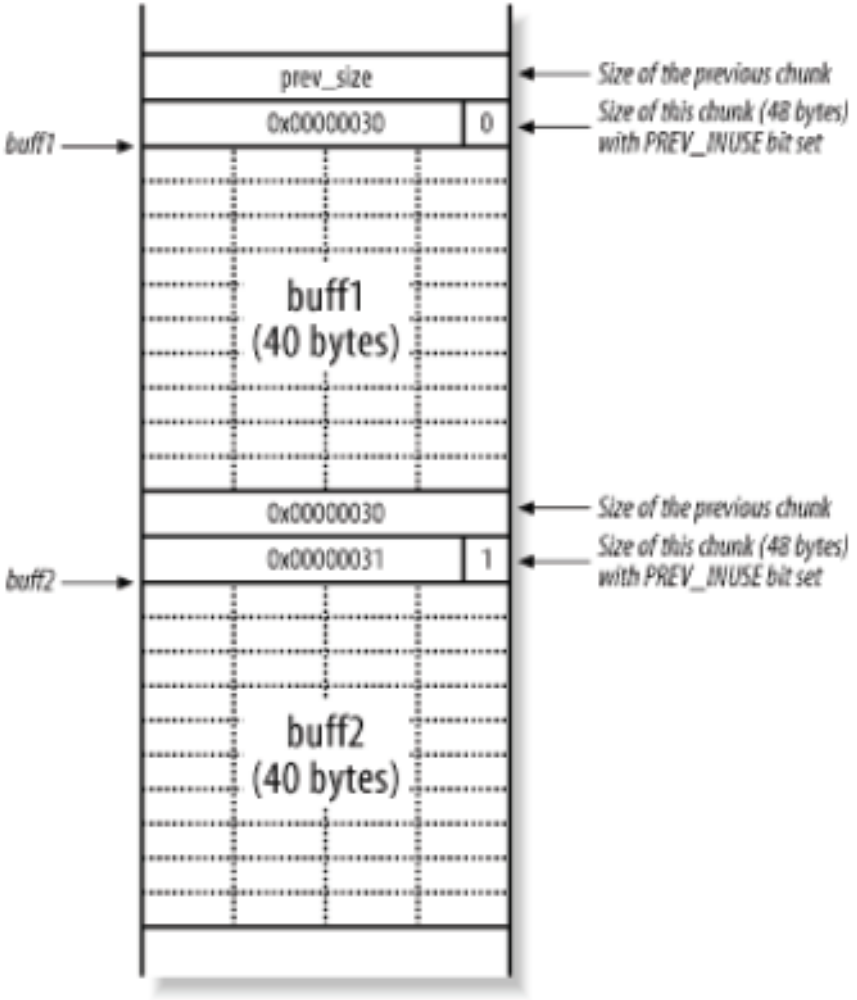



**Hotovo?**

- Útočník nepotrebuje prepísať návratovú adresu
  - Lokálne premenné môžu tiež ovplyvniť beh programu
  - Premenné v podmienkach
  - Smerníky na funkcie

# Heap overflow

- Pretečenie môže nastať aj na heape
- Útok:
  - Prepíš heap cieľovou adresou
  - Dúfaj, že obeť použije prepísaný odkaz na funkciu



# Heap overflow

- Ochrana:
  - Môžu sa použiť kanáriky, ale je to ťažké urobiť efektívne
  - Skontrolovať veľkosť buffera pred samotným zápisom.
    - Musí sa to urobiť pred každou funkciou zapisujúcou do buffera

# Non-eXecutable pamäť (NX / W⊕X)

- Rozdeľ pamäť na
  - Executable (na ukladanie kódu)
  - Non-executable (na ukladanie dát)
- A processor zabráni vykonať non-executable kód
  - Toto sa môže urobiť pre zásobník, alebo akúkoľvek stránku pamäte
- Útočník nemôže skákať do svojho kódu, keďže bude označený za non-executable
- Moderné CPU poskytujú pre to hardwareovú podporu

# Return-to-libc útok

- Cesta ako obísť non-executable pamäť
  - Využiť buffer overflow na skok do kódu, ktorý tam už je, hlave do kódu v knižnici libc
- Libc je bohatá systémová knižnica poskytujúca veľa možností pre útočníka: system, exec, fork
- Veľa knižníc, vrátane libc poskytuje dostatok operácii aby boli Turingovsky úplne.

# Control Flow Integrity (CFI)

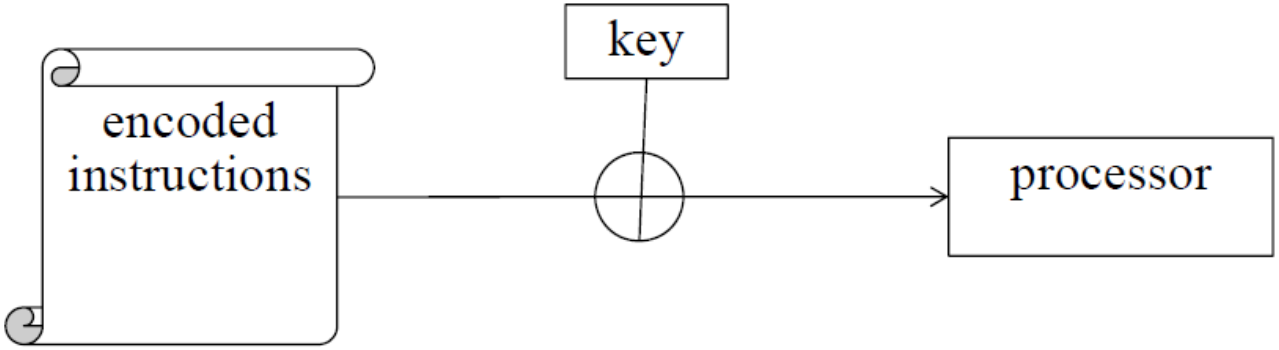
- Return-to-libc útok môže byť odhalený, keďže väčšinou sa jedná o neobvyklé volanie
  - Napr. funkcia foo() nikdy nevolá rutinu bar(), bar() nie je ani v kóde funkcie foo(). Avšak počas behu foo() na škodlivom kóde dôjde k zavolaniu bar()
- Return-to-libc útok môže byť zablokovaný, keďže také nezvyčajné volania môžu byť počas runtime detekované.
  - Avšak má to zvýšené administratívne nároky

# Address space layout randomisation (ASLR)

- Útočník potrebuje detailné informácie o rozložení pamäti
- Znáhodnením rozloženia pamäti útok značne skomplikujeme.
  - Napr. posunieme začiatok heapu / zásobníka o nejakú náhodnú hodnotu
- Kedy znáhodňovať?
  - Keď spustíme program?
  - Pri vytvorení nového vlákna (fork())?

# Znáhodnenie inštrukčnej sady

- Pre ešte väčšiu komplikáciu útoku:
  - Zakódovať inštrukčnú sadu, rôzne pre každý process




- Nevyhnutná HW podpora, aby to bolo efektívne
- Útočník nevie napísať kód, keďže nevie ako zakódovať požadované inštrukcie.

# Dynamická ochrana (rekapitulácia)

- Kanáriky
- non-executable pamäť
- Address space layout randomisation (ASLR)
- Instruction set randomisation

- Žiadna z týchto ochrán nie je dokonalá
- Šikovný útočník môže a nájde cestu ako ich obísť

# Buffer overflow - zhrnutie

- Buffer overflow chyby patria medzi najčastejšie zraniteľnosti
- Akýkoľvek C(++) kód pracujúci na nedôveryhodnom vstupe je ohrozený resp. akýkoľvek C(++) kód je ohrozený
- Obrana voči buffer overflow chybám je ťažká
  - Stále prebieha súboj medzi obrannými mechanizmami a novými typmi útokov

# Buffer overflow

- Buffer overflow súvisí s troma všeobecnejšími problémami
- Absencia validácie vstupu
- Mixovanie dát a kódu
- Spoliehanie sa na abstrakciu, ktorá nie je 100% garantovaná a dodržiavaná
  - Napr. typy a rozhranie procedúr v C

```
int f(float f, boolean b, char* buf);
```

# SYSTÉMOVÉ ZDROJE

# Systémové zdroje

- Programy často potrebujú prístup k rôznym zdrojom
  - Knižnice, nastavenia, environment premenné, súbory, ...
- Útočník môže ovplyvniť mechanizmy na prístup k týmto zdrojom a kompromitovať tak program
  - Čiže je potrebné takýmto útokom zabrániť

# Namespace

- Klient (proces) požiada o prístup k zdroju (súbor) od systému (OS) pomocou **mena**
- Systém transformuje **meno** na zdroj pomocou previazania na namespace
  - Mapovanie medzi názvom a zdrojom
  - Napr. cesta k súboru na súbor  / adresár
- Namespace sa používa na veľa miestach
  - Andoid Intents
  - URL
  - DNS
  - ...

# Namespace resolution útoky

- Útočník si **volí názov**
  - Použije vhodne zvolený názov, ktorým prekabáti parser a dostane tak prístup k zdroju
  - Upraví spôsob konštrukcie mena (napr. Environment premenné) a presmeruje tak obeť na škodlivý zdroj
- Útočník má kontrolu nad **namespace mapovaním**
  - Vytvorí linku a presmeruje obeť na škodlivý zdroj
- Útočník má prístup ku zdroju
  - Obeť môže považovať daný zdroj za bezpečný, aj keď k nemu má útočník prístup

# Útočník si **volí názov**

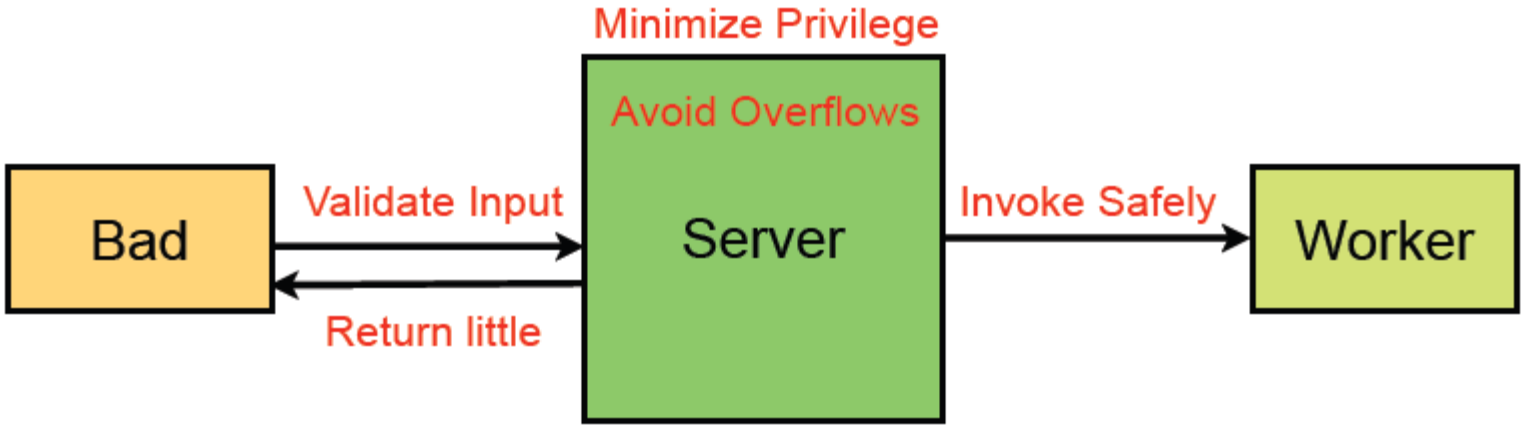
- Viacero spôsobov ako pomenovať to isté
  - Súbory: /x/data alebo /y/z/../../x/data alebo /y/z/%2e%2e/x/data
  - Podobne v URL, DNS, ...
- Umožní útočníkovi prístup k inak pre neho neprístupnému zdroju
- Okabátiť proces, aby načítal nedôveryhodný súbor
  - E.g. škodlivý PHP súbor
  - **File inclusion útok**

# Search Path zraniteľnosť

- Útočník môže podvrhnúť obeti zlý zdroj pomocou „search path“ environment premennej
- Keď program potrebuje knižnicu
  - Linker vyhľadá súbor v LD_PATH adresároch
    - Môže obsahovať aj aktuálny adresár
- Útok:
  - Útočník do home adresára uloží škodlivú knižnicu
  - Naštartuje privilegovaný program z domovského adresára
  - Linker načíta škodlivú knižnicu

# Bezpečné programovanie

- Validácia vstupu.
- Vyhýbať sa buffer overflow chybám
- Minimalizovať privilégia procesu
- Obozretne volať / pristupovať k iným zdrojom
- Obozretne posielať spätnú informáciu



# Bezpečné programovanie