Typy chyb a jejich hledání v jazyce C

1. **Syntaxové chyby:**
   Příklad:

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

Tento kód má syntaxovou chybu ve formě chybějícího zahrnutí hlavičkového souboru ¡stdio.h¿. Kompilátor vygeneruje chybovou zprávu, která upozorní na tento problém.

Hledání: Syntaxové chyby jsou obvykle detekovány kompilátorem, který vydá chybové zprávy při kompilaci kódu. Stačí procházet chybové zprávy, abyste našli a opravili syntaxové chyby.

2. **Logické chyby:**
   Příklad:

```
int sum(int a, int b) {
return a - b; // Chybně odečítáme místo sčítáme
}
```

Tento kód má logickou chybu, kde funkce `sum` odečítá `b` od `a` místo aby je sčítala.

Hledání: Logické chyby jsou obtížnější najít, protože program kompiluje a spouští se bez chybějících chybových zpráv. K hledání logických chyb lze použít ladící nástroje, jako je GDB, a testování vstupů programu.

3. **Chyby za běhu programu:**
   Příklad:

```
int main() {
int a = 10, b = 0;
int result = a / b; // Dělení nulou
printf("Result: %d\n", result);
return 0;
}
```

Tento kód má chybu za běhu programu, protože dělíme `a` nulou, což způsobí dělení nulou.

Hledání: Chyby za běhu programu mohou být detekovány pomocí ladících nástrojů, jako je GDB. Ladění umožňuje programátorovi sledovat chování programu v průběhu jeho běhu a identifikovat problémové části kódu.

4. **Chyby přístupu do paměti:**
   Příklad:

```
int main() {
int arr[5] = {1, 2, 3, 4, 5};
printf("%d\n", arr[10]); // Přístup mimo rozsah pole
return 0;
}
```

Tento kód má chybu přístupu do paměti, protože se pokoušíme přistoupit k prvkům pole mimo jeho rozsah.

Hledání: Chyby přístupu do paměti mohou být obtížné najít, ale lze je identifikovat pomocí nástrojů pro sledování paměti, jako je Valgrind. Tyto nástroje sledují přístupy do paměti a upozorňují na neplatné operace.

5. **Chyby ve správě paměti:**

Příklad:

```
int main() {
int *ptr = malloc(10 * sizeof(int));
ptr = NULL; // Ztráta odkazu na alokovanou paměť
free(ptr); // Pokus o uvolnění nulového ukazatele
return 0;
}
```

Tento kód má chyby ve správě paměti: ztrátu odkazu na alokovanou paměť a pokus o uvolnění nulového ukazatele.

Hledání: Chyby ve správě paměti mohou být identifikovány pomocí nástrojů pro sledování paměti, jako je Valgrind nebo AddressSanitizer. Tyto nástroje sledují alokaci a uvolňování paměti a upozorňují na potenciální problémy.

Časté chyby

Pro snažší identifikaci programovacích defektů je praktické mít seznam s jednoznačnými identifikátory. Podobně jako pro bezpečnostní problémy existuje výčet Common Vulnerabilities and Exposures (CVE)[1], vznikl i seznam Common Weakness Enumeration (CWE)[2] spravovaný Mitre Corporation. Tento seznam obsahuje jak možné problémy v implementaci počítačových programů, tak i jejich návrhu a architektuře. Každému defektu je věnována stránka s popisem problému a určením, ve které části vzniku softwaru se chyba objevuje. Dále následuje výčet možných důsledků, příklady chybných programů s vysvětlením, co a proč je v nich špatně, a návrhy, jak tomuto problému předcházet, případně jak ho odhalit. Všechny stránky na webu CWE jsou řazeny do stromové hierarchie, takže je možné se snadno z popisu jednoho defektu dostat k podobným chybám, případně obecnému popisu skupiny chyb.

Samotný web by mohl v budoucnu sloužit nejen jako databáze znalostí o programovacích chybách. Jeho potenciál je větší. Pro usnadnění práce s nástroji pro statickou analýzu by bylo velmi vhodné standardizovat sadu anotací, pomocí kterých by bylo možné vyznačit ve zdrojovém kódu problematické úseky a tím dát najevo, že daný řádek kódu nebo funkce opravdu má vypadat tak, jak vypadá, a nejde o chybu.

V následujících odstavcích jsou představeny některé z možných problémů.

[1] http://cve.mitre.org/

[2] http://cwe.mitre.org/

Použití neinicializované proměnné – CWE-457

V jazyku C i C++ se nově deklarovaným proměnným automaticky nepřiřazuje žádná výchozí hodnota. Obsahují tedy to, co bylo v paměti na jejich adrese uloženo dřív. Tato data jsou jen velmi vzácně k něčemu užitečná a je tedy praktické co nejdříve proměnným přiřadit nějakou jasně definovanou hodnotu. Neinicializované proměnné mohou vést k pádu programu. Pokud by se potenciálnímu útočníkovi podařilo zajistit, aby na adrese neinicializované proměnné byla jím připravená data, mohl by ovlivnit běh programu.

Příklad:

```
int x; // neinicializovaná proměnná
printf("Hodnota x: \%d\\n", x); // může obsahovat nedefinovanou hodnotu
```

Dereference ukazatele NULL – CWE-476

Dereference ukazatele NULL obvykle vede k pádu programu se signálem SIGSEGV. K této chybě může dojít jak souběhem vláken se špatnou synchronizací, tak i prostým opomenutím programátora. Obvykle se chyba projevuje v částech kódu, které se nepoužívají příliš často a tedy mohou snadněji projít testováním neodhaleny.

```
int *ptr = NULL;
*ptr = 5; // Dereference NULL pointer
```

Únik paměti – CWE-401

Pokud program korektně nesleduje a neuvolňuje alokovanou paměť, může docházet k postupnému zvyšování spotřeby paměti. To vede k nižší spolehlivosti programu.

```
void function() {
int *ptr = malloc(sizeof(int));
// paměť není uvolněna, dochází k úniku paměti
}
```

Únik deskriptorů – CWE-775

Podobný případ neuvolňování paměti je i situace, kdy program otevře např. soubor, ale po skončení práce s ním ho už nezavře. Přestože po skončení programu operační systém všechny otevřené soubory zavře, v případě větších a déle běžících systémů může snadno dojít k vyčerpání všech dostupných deskriptorů a následnému selhání při otevírání dalšího souboru.

Příklad:

```
FILE *fp = fopen("file.txt", "r");
// neprovádí se fclose(fp), což vede k úniku deskriptoru
```

Použití paměti po uvolnění – CWE-416

Použití paměti, která byla uvolněna, může vést k pádu programu, použití neočekávané hodnoty nebo i spuštění libovolného kódu. Při přístupu k uvolněné paměti může dojít k poškození dat, pokud už byla stejná paměť alokována jinde.

Příklad:

```
int *ptr = malloc(sizeof(int));
free(ptr);
*ptr = 5; // použití uvolněné paměti
```

Mrtvý kód – CWE-561

Mrtvý kód (dead code) je takový kód, který se nikdy nevykoná. Přestože se technicky nejedná o chybu a na běh programu nemá vliv, může takový kód komplikovat údržbu programu a tím brzdit opravu jiných, už závažnějších chyb. Důvodem, proč se některá část kódu nikdy neprovede, může být například testování podmínky, která je vždy pravdivá (resp. nepravdivá).

Příklad:

```
void function(int x) {
if (x > 0) {
// mrtvý kód, protože x nikdy nebude záporné
} else {
// tento blok kódu se vždy vykoná
}
}
```

Race condition – CWE-362

Race condition je situace, kdy výsledek nějaké operace je závislý na pořadí a načasování jednotlivých operací vícevláknových programů. Souběh obvykle bývá způsoben chybným přístupem ke sdílenému prostředku, např. modifikací datové struktury z více vláken zároveň.

Příklad:

```
int counter = 0;

void increment_counter() {
counter++;
}

// vlákno 1
increment_counter();

// vlákno 2
```

```
increment_counter();

// Výsledek není deterministický kvůli nedostatečné synchronizaci
```

Chyby zámků – CWE-667

Při používání sdíleného zdroje více vlákny programu je obvykle třeba zajistit, aby zdroj mohl být v jednom okamžiku používán pouze jedním vláknem. K tomu je mimo jiné možné používat zámky, mutexy a semafory. Existuje mnoho různých konkrétních chyb při práci se zámky, jako je zapomenutí nutnosti získat zámek, neošetření selhání synchronizační funkce, deadlock atd.

Příklad:

```
#include <pthread.h>

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *thread_func(void *arg) {
pthread_mutex_lock(&lock);
// zde může být kód, který manipuluje se sdílenými daty
pthread_mutex_unlock(&lock);
return NULL;
}
```

Přetečení bufferu – CWE-120

K přetečení bufferu dochází tehdy, když program zkopíruje data ze vstupního bufferu do výstupního, aniž by zkontroloval, že velikost vstupního bufferu není větší než velikost výstupního bufferu.

Příklad:

```
#include <string.h>

void copy_string(char *src) {
char dst[10];
strcpy(dst, src); // Přetečení bufferu muze nastat, pokud src obsahuje více nez 10 znaku
}
```