Učebný text k prednáške
Bezpečné programovanie 2

Richard Ostertág, KI FMFI UK Bratislava 2020

Materiál je výstupom Rozvojového projektu Univerzity Komenského a Ministerstva školstva, vedy, výskumu a športu SR č. 002UK-2-1/2018 – „Vzdelávanie pre informačnú spoločnosť“ v oblasti Podpora vysokých škôl pri plnení záväzkov prijatých v rámci Národnej koalície pre digitálne zručnosti a povolania SR.

Jazyková úprava:

Rukopis neprešiel jazykovou úpravou.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

Cieľom tohto materiálu je popísať ďalšie typy zraniteľností, s ktorými sa stretávame pri vývoji aplikácií. Pri jeho písaní sme vychádzali okrem vlastných skúseností aj zo zdrojov [1–10], ktoré odporúčame aj čitateľovi, ktorý ma záujem o hlbšie samoštúdium.

Budeme sa venovať problematike dočasných súborov a niektorým ďalším známejším problémom. Konkrétne si ukážeme zneužitie pretečením vyrovnávacej pamäte („buffer overflow“) a tiež si vysvetlíme, ako funguje útok vložením kódu („code injection“). Potom si spomenieme niektoré špecifiká programovacích jazykov, ako modifikátor `final` v jazyku Java a jeho význam pre bezpečné programovanie. Ďalej sa budeme venovať menej známym štruktúrovaným výnimkám a bezpečnostným implikáciám použitia ich filtra. Spomenieme aj problémy s časovou závislosťou pri uvoľňovaní zdrojov.

Ani práca s celými alebo desatinnými číslami nie je bez „temných kútov“, kde na programátora môže striehnuť rôzne nečakané správanie. Materiál ukončíme povestnou čerešničkou na torte, ktorou sú v tomto prípade bezpečnostné aspekty optimalizujúcich kompilátorov. Aj keď sú optimalizácie dobre mienené (s cieľom generovať čo najkratší a najrýchlejší kód), ukážeme si, že môžu mať nečakané dopady na bezpečnosť aplikácie.

Kapitola 2

Bezpečné programovanie 2

Aplikácie často potrebujú používať dočasné súbory. Môžu si v nich pamätať medzivýsledky výpočtov alebo napríklad si do nich odkladať aktuálne rozpracovaný dokument ešte pred tým, než je prvýkrát používateľom uložený na disk. Nasledujúca časť je venovaná niektorým bezpečnostným zraniteľnostiam, ktoré súvisia s použitím dočasných súborov.

2.1 Dočasné súbory

Dočasné súbory sa obvykle ukladajú do adresárov, ktoré sú prístupné všetkým používateľom na čítanie aj na zápis. Takáto situácia je typická hlavne pri centrálnom úložisku dočasných súborov, keďže každý používateľ môže spúšťať program, ktorý bude dočasné súbory potrebovať. Obvyklými adresármi pre takéto súbory sú napríklad:

- `/tmp` (Linux)
- `/var/tmp` (Linux)
- `C:\Windows\TEMP` (Windows – centrálne úložisko)
- `C:\Users\Name\AppData\Local\Temp` (Windows – lokálne úložisko)

Dočasné súbory by mali byť zmazané aplikáciou hneď, keď ich už nepotrebuje. Šetrí sa tým diskový priestor a znižuje sa šanca, že v budúcnosti nastane kolízia. Často však programátori na to zabúdajú, a tak sa ich aj niektoré systémy snažia upratovať. Môžu ich zmazať hneď, ako aplikácia skončí alebo napríklad pri štarte alebo počas vypínania systému, keď je jasné, že už žiadna aplikácia nebeží, a teda dočasné súbory už nikto nepotrebuje. Pokiaľ je systém prevádzkovaný nepretržito (napríklad server) je možné dočasné súbory upratovať napríklad raz za deň.

Žiaľ, opustené dočasné súbory nie sú zriedkavé a nie každý operačný systém má zabudované riešenie na ich upratovanie. Preto existujú rôzne nástroje na čistenie adresárov s dočasnými súbormi. Tieto nástroje môžu byť od tretích strán alebo vytvorené lokálnym

administrátorom (napríklad ako cron job, ktorý pravidelne odstraňuje niekoľko dní staré dočasné súbory). Tieto nástroje sú tiež náchylné na útoky. Niekedy bežia aj pod právami roota. Ak útočník nahradí dočasný súbor symbolickou linkou na iný súbor, alebo priamo v dočasnom adresári takú linku vytvorí, tak tieto nástroje (ak sú nevhodne naprogramované) môžu zmazať aj prilinkované súbory.

Je dôležité, aby dočasné súbory mali nepredpovedateľné meno, pretože inak sa dajú zneužiť na rôzne útoky. Ak napríklad vieme odhadnúť, aké dočasné súbory bude vytvárať program, ktorý beží s právami roota, tak ho môže útočník zneužiť na prepísanie ľubovoľného súboru na disku. Stačí, ak vytvorí v `/tmp` adresári symbolickú linku s predpovedaným menom na chránený súbor. Keď sa privilegovaný program pokúsi do tohto súboru zapisovať, tak vlastne prepíše útočníkom nalinkovaný súbor. Podobne môže neprivilegovaný program prepísať ľubovoľný súbor používateľa, ktorý tento program spustil. Stačí, ak útočník vytvorí v `/tmp` adresári symbolickú linku s predpovedaným menom na používateľov súbor. Tým zabezpečí, že si používateľ po spustení programu sám prepíše súbor, ku ktorému by inak utočník ani nemal prístup.

Funkcie ako `char *tmpnam(char *s)` alebo `char *mktemp(char *template)` sa snažia vytvoriť náhodný a unikátny názov dočasného súboru. Ich použitie môže byť nasledovné:

```
1 if (tmpnam(filename)) {
2   tmpfile = fopen(filename, "bw+");
3   ...
4 }
```

Problémom je však čas, ktorý prebehne medzi kontrolou unikátnosti na riadku 1 a časom, kedy je súbor na riadku 2 naozaj vytvorený, pretože v tomto čase môže niekto získané meno obsadiť. Po anglicky sa tento problém zvykne nazývať „Time of Check to Time of Use“ v skratke TOCTOU. Pre vyriešenie tejto časovej závislosti je potrebná podpora operačného systému. Vyhnúť sa problémom s TOCTOU je možné použitím iných, vhodnejších funkcií, ktoré rovno vracajú otvorený súbor a časovú závislosť riešia s využitím podpory OS:

- `FILE *tmpfile(void)`
  Funkcia `tmpfile()` otvorí jedinečný dočasný súbor v binárnom read/write (bw+) móde. Súbor bude operačným systémom automaticky vymazaný, keď bude zatvorený alebo keď bude program ukončený.

- `int mkstemp(char *template)`
  Funkcia `mkstemp()` vracia popisovač (descriptor) otvoreného dočasného súboru, ktorého meno odvodí zo zadanej reťazcovej šablóny `template`. Posledných šesť znakov šablóny musí byť reťazec `"XXXXXX"`. Tieto znaky sú pri zavolaní funkcie nahradené reťazcom, ktorý zabezpečí unikátnosť výsledného názvu súboru. Pretože bude táto šablóna funkciou upravená, nesmie ísť o reťazcovú konštantu, ale mala by byť deklarovaná ako pole znakov.

Dočasné súbory musia byť otvorené s exkluzívnym prístupom a vhodnými prístupovými právami. Napríklad s oprávneniami 600, čo znamená prístup na čítanie a zapisovanie iba pre vlastníka súboru. Na nasledovnom programe si ukážeme odporúčaný spôsob práce s dočasnými súbormi:

```
1  char sfn[15] = "/tmp/ed.XXXXXX";
2  FILE *sfp; int fd = -1;
3  if ((fd = mkstemp(sfn)) == -1 || (sfp = fdopen(fd, "w+")) == NULL) {
4    if (fd != -1) {
5      unlink(sfn); close(fd);
6    }
7
8    /* ošetrenie chyby */
9  } else {
10   unlink(sfn); /* okamžite vymazať z adresára */
11
12   /* použitie dočasného súboru */
13
14   close(fd);
15 }
```

Pokiaľ existuje proces, ktorý má súbor otvorený, tak funkcia `unlink()` odstráni súbor z adresára, ale fyzicky ho operačný systém vymaže, až keď ho všetky procesy zavrú. Toto riešenie zabezpečí, že aj keď bude program predčasne ukončený (napríklad príkazom `kill`), tak dočasný súbor bude aj tak vymazaný. Navyše dočasný súbor je v adresári pre dočasné súbory viditeľný len minimálny čas, čím sa znižuje šanca kolízie, ako aj útoku.

2.2 Pretečenie vyrovnávacej pamäte

Pretečenie vyrovnávacej pamäte (buffer overflow) sa vyskytuje, keď program načítava dáta do vyrovnávacej pamäte (buffera) a pritom nekontroluje množstvo zapísaných údajov. Programátor často odhadne maximálnu veľkosť načítavaných údajov a ticho dúfa, že v praxi nikdy nebude na vstupe viac údajov. Pri bežnom používaní aplikácie to aj môže byť pravda.

Napríklad používateľské meno nebude asi nikdy dlhšie ako 50 znakov. Preto programátor s rezervou zvolí 100 B ako veľkosť vyrovnávacej pamäte pre načítanie používateľského mena.

Ak však útočník zistí ignorovanie kontroly dĺžky vstupu, tak môže aplikácii úmyselne poslať neprimerane dlhý vstup. Pre útočníka nie je problém zadať meno, ktoré má aj 200 znakov. Pokiaľ program naozaj nekontroluje dĺžku vstupu a cielene sa neobmedzuje na načítanie maximálne 99 znakov[1], tak sa pri načítaní dlhého vstupu prepíše celá vyrovnávacia pamäť a aj údaje, čo sú v pamäti uložené za ňou.

Útok si prakticky vyskúšame. Najprv musíme vytvoriť testovací zraniteľný program, ktorý uložíme do súboru s názvom `attack.c`:

[1] Jeden znak treba nechať aj na ukončenie reťazca. Toto je ďalšia častá chyba.

```c
#include <stdio.h>

const char* password="SuperSecretPassword123";

struct {
  char buf[100];
  char* name;
} user;

void main(void)
{
  user.name = user.buf;
  printf("Enter user name: ");
  scanf("%s", user.name);

  printf("%s\n", "... processing user name ...");

  printf("%s\n", user.name);
}
```

Po uložení súboru môžeme program skompilovať nasledovným príkazom:

```
$ gcc -no-pie attack.c
```

Parameter `-no-pie` hovorí kompilátoru, aby nevytvoril tzv. „position independent“ kód. Tým sa zjednoduší a zafixuje rozloženie dát a programu v pamäti, čo nám zjednoduší útok. Bez tohto parametra by útok bol tiež možný, ale zbytočne by sa pre tieto demonštračné účely komplikoval. Kompiláciou sme vytvorili spustiteľný program, ktorý kompilátor uložil do súboru `a.out`. Teraz použijeme program `objdump` pre získanie informácie o uložení premenných v programe:

```
$ objdump -x a.out | grep password
0000000000601040 g     O .data  0000000000000008   password
```

Takto sme získali umiestnenie globálnej premennej `password` v dátovom segmente. Na tomto mieste sa nachádza hodnota uložená v tejto premennej. Je to ale iba ukazovateľ na reťazec s heslom. Preto na adrese premennej `password` (0x601040) je iba adresa, kde je uložené hľadané heslo.

```
$ objdump -s -j .data a.out

a.out:     file format elf64-x86-64

Contents of section .data:
 601030 00000000 00000000 00000000 00000000  ................
 601040 94064000 00000000                    ..@.....
```

Z výpisu vidíme, že tajný reťazec je uložený na 64-bitovej adrese: 0x00000000 00400694 (uloženej od najmenej významných bajtov po najvýznamnejšie). Ak budeme hypoteticky

predpokladať, že program beží na vzdialenom serveri a my mu vieme len poslať vstup na výzvu „Enter user name:“, tak vyzbrojený vyššie získanou informáciou môžeme pristúpiť k vzdialenému zneužitiu zraniteľnosti v programe `a.out`:

```
1 $ perl -e 'print "a"x104; print "\x94\x06\x40"; print "\x00"x5' ↩
   ↪| ./a.out
2 Enter user name: ... processing user name ...
3 SuperSecretPassword123
```

Vidíme, že program nám po zadaní nami vhodne skonštruovaného vstupu sám vypísal tajné heslo. Štruktúra „magického“ vstupu je nasledovná:

1. Sto znakov dlhý reťazec áčiek, ktorým sa zaplní vyrovnávacia pamäť `buf`.
2. Štyri áčka sa ešte pridajú na preklenutie zarovnania adries položiek v dátovej štruktúre `user` na násobok ôsmych (tzv. padding).
3. Nakoniec nasleduje 64-bitová adresa 0x00000000 00400694 reťazca s tajným heslom, ktorá prepíše premennú `name` v štruktúre `user`.

Práve vďaka 3. kroku nakoniec program sám vypíše tajné heslo, keď vykoná riadok 19. Premenná `user.name` v tomto prípade vďaka pretečeniu vyrovnávacej pamäte bude ukazovať na tajné heslo.

2.3 Vkladanie kódu

Vkladanie kódu / riadiacich dát (code injection) je ďalším známym typom útoku. Jeho myšlienka spočíva v tom, že každý programovací jazyk má svoju špecifickú syntax a špecifické riadiace znaky. Ak sa neošetrené dáta od klienta vložia do reťazca, v ktorom sa konštruuje príkaz v nejakom programovacom jazyku, tak tieto vložené dáta môžu zmeniť štruktúru a interpretáciu vytvoreného príkazu.

Pri vývoji aplikácií sa často stretávame s použitím viacerých programovacích jazykov pri ich implementácii. Typickým príkladom sú webové aplikácie, kde sa bežne súčasne používajú jazyky ako napríklad SQL, HTML, JavaScript, XML, HTTP. O to ťažšie je pre programátora ošetriť vstup tak, aby nemohol byť pri spracovaní v aplikácii zneužitý. Podľa toho, kde sa útočníkovi podarí vložiť svoj kód, hovoríme o rôznych konkrétnych typoch útokov (ktoré ale všetky spadajú do jednej rodiny vkladania kódu):

- SQL injection
  Ide o útok, keď sa útočníkovi podarí vložením vhodných údajov zmeniť význam SQL príkazu. Podrobnejšie si ho popíšeme v nasledovnej podčasti.
- XPath injection
  Je podobný útok ako SQL injection, ale na jazyk XPath, ktorým sa popisuje vyhľadávanie údajov v XML súboroch.

- HTML injection (markup injection)
  Tento útok umožní útočníkovi zmeniť význam HTML dokumentu. Konkrétny príklad takéhoto všeobecného útoku je napríklad „cross-site request forgery“ známy aj pod skratkou CSRF alebo XSRF.

- JavaScript injection
  V tomto prípade sa podarí útočníkovi vložiť do stránky kód v jazyku JavaScript. Konkrétny príklad takéhoto všeobecného útoku je napríklad „cross-site scripting“ známy aj pod skratkou XSS.

2.3.1 SQL injection

Predstavme si, že máme nasledovný príkaz v jazyku SQL:

```
1 select * from Pouzivatelia where Meno = 'Janko Hrasko';
```

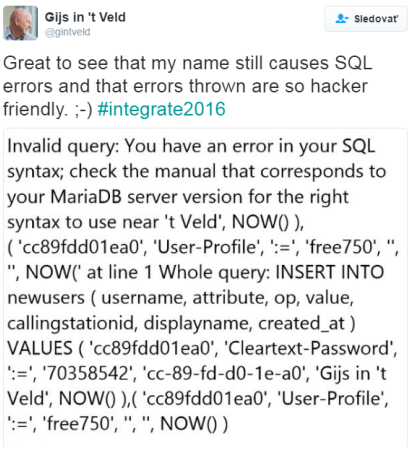
Tento príkaz vráti zoznam všetkých používateľov z tabuľky `Pouzivatelia`, ktorých meno je „Janko Hrasko“. Predpokladajme, že meno používateľa ja zadané samotným používateľom a vkladá sa do príkazu medzi apostrofy. Výsledný reťazec sa potom pošle SQL serveru na interpretáciu. Ak používateľ zadá vstup ’ or ’’ = ’, tak výsledný reťazec s SQL príkazom bude:

```
1 select * from Pouzivatelia where Meno = '' or '' = '';
```

Tento príkaz SQL server pochopí tak, že vráti zoznam všetkých používateľov z tabuľky `Pouzivatelia`, ktorí majú meno prázdny reťazec (taký nebude asi žiadny) alebo pre nich platí, že prázdny reťazec je prázdny reťazec (to budú všetci). Útočník takýmto spôsobom môže získať zoznam všetkých používateľov.

Ak útočník na vstupe zadá ako svoje meno ’ and 1 = 0 union all select * from Tabulka; --, tak tým zmení príkaz nasledovne:

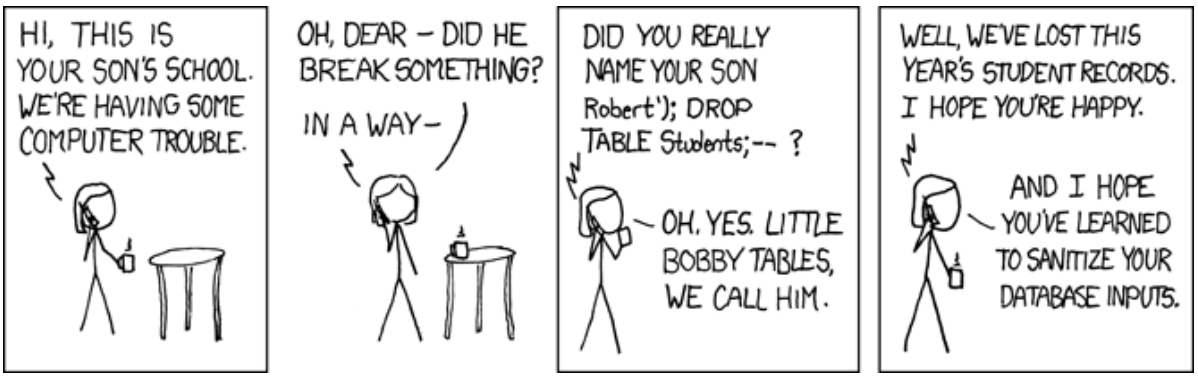
```
1 select * from Pouzivatelia where Meno = '' and 1 = 0 union all
    select * from Tabulka; --';
```



Takýto reťazec SQL server pochopí ako dva príkazy `select`, ktorých výsledky má spojiť do jedného spoločného výsledku. Pritom prvý `select` vráti zoznam všetkých používateľov z tabuľky `Pouzivatelia`, ktorých meno je prázdny reťazec (taký asi nebude žiadny) a zároveň pre nich platí že `1 = 0` (taký nebude určite žiadny). Prvý select teda nevráti žiadny výsledok. Druhý `select` vráti kompletný obsah tabuľky `Tabulka`. Znaky `--` predstavujú v jazyku SQL začiatok komentára (až do konca riadku). V konečnom dôsledku môže útočník zadaním takéhoto mena získať kompletný obsah ľubovoľnej tabuľky v databáze.

Niekedy podobná chyba vznikne aj keď sa používateľ nesnaží na systém útočiť, ale len zadá svoje meno (Gijs in ’t Veld), ktoré obsahuje programátorom neočakávaný znak (ako vidno na obrázku na okraji).

Randall Munroe na obrázku 2.1 dobre a vtipne vystihol myšlienku SQL injection:



Obr. 2.1: Exploits of a Mom (zdroj: https://xkcd.com/327/)

Nie vždy aplikácia vráti útočníkovi celý výsledok SQL príkazu. Niekedy vráti iba časť výsledku. Dokonca môže vracať informáciu iba nepriamo, napríklad v podobe chybového hlásenia alebo rôznej dĺžky spracovania požiadavky. V takomto prípade hovoríme o tzv. „blind sql injection“.

Predpokladajme, že útočník chce zistiť výšku výplaty svojho kolegu. Podarilo sa mu spraviť „blind sql injection“, ale výsledok vie odhadnúť len podľa dĺžky spracovania SQL príkazu. Pre zistenie výšky platu použije algoritmus binárneho vyhľadávania. Pritom potrebuje porovnať skutočný plat s odhadovaným. Príslušný SQL príkaz zostaví tak, že pokiaľ je skutočný plat menší alebo rovný ako odhadovaný, tak príkaz skončí rýchlo, ale ak je väčší tak bude trvať dlho. Potom podľa dĺžky odozvy vie usúdiť, ktorú časť intervalu má ďalej prehľadávať.

„SQL injection“ môžeme riešiť pomocou univerzálnej ochrany pred akýmkoľvek vkladaním kódu — správnym ošetrením špeciálnych znakov. „Prepared queries“ sú však špecifickým spôsobom ochrany pred týmto druhom útoku. Najprv pošleme na server príkaz so zástupnými znakmi namiesto parametrov. Až keď server príkaz sparsuje, tak ho pred vykonaním doplníme o požadované parametre, ktoré už ale nemôžu nijak zmeniť jeho význam.

2.4 Java – kľúčové slovo final

Programovací jazyk Java má kľúčové slovo **`final`**, ktoré podľa kontextu, v ktorom je použité, môže mať niekoľko významov:

- **`final double`** `PI = 3.1415926;`
  Do premennej, ktorá je **`final`** je možné priradiť iba raz, najčastejšie hneď pri jej deklarácii. Po inicializovaní premennej jej hodnotu už nie je možné meniť.

- **`final void`** `m() ...`
  Metóda, ktorá je **`final`**, nemôže byť predefinovaná (overridden) v potomkoch.

- `final class T ...`
  Od triedy, ktorá je `final`, nie je možné dediť a tak vytvárať jej potomkov.

Z vyššie uvedeného teda vyplýva, že ak trieda `T`, ani metóda `m` nie sú `final`, tak môžeme vytvoriť novú triedu `N`, ktoré bude potomkom triedy `T` a bude mať predefinovanú metódu `m`.

Prečo by nám niečo takéto mohlo (z bezpečnostného hľadiska) pri niektorých triedach vadiť? Treba si uvedomiť, že všetky metódy v jazyku Java sú virtuálne (na rozdiel od jazyka C++). To znamená, že ak v premennej `t` typu `T` máme vďaka polymorfizmu v skutočnosti uloženú inštanciu triedy `N` (ktorá je potomkom triedy `T`), tak zavolaním `t.m()` sa nezavolá metóda `T.m`, ale sa zavolá metóda `N.m`. A to dokonca aj v kóde, ktorý bol skompilovaný ešte pred vytvorením samotnej triedy `N`.

Niektoré štandardné triedy majú pre správne fungovanie Java aplikácie zásadný význam. Napríklad trieda `String` je implementovaná ako immutable. Ak by jej potomkovia túto vlastnosť pri svojej implementácii nezachovali[2], viedlo by to k neočakávanému správaniu. Pokiaľ sa pri viac-vláknových aplikáciách zdieľajú immutable objekty, tak zmena tejto vlastnosti pri potomkoch je vážny problém.

Rovnako očakávame, že funkcia `int compareTo(String anotherString)` pre porovnanie dvoch reťazcov funguje správne. Útočník by napríklad mohol vytvoriť potomka triedy `String` s predefinovanou implementáciou `compareTo`, ktorá by vždy vrátila hodnotu 0 (reťazce sa rovnajú). Ak by potom zadal do funkcie overujúcej heslo akýkoľvek reťazec s predefinovanou implementáciou `compareTo`, tak by, v závislosti na jej implementácii, mohlo byť toto heslo považované za správne. Aj z týchto dôvodov je trieda `String` v Jave `final`. Ako vidíme, tak na jednej strane je `final` trieda bezpečnejšia, na druhej strane však zas prichádzame o možnosť rozšíriteľnosti.

2.5 Štruktúrované spracovanie výnimiek

Štruktúrované spracovanie výnimiek (Structured Exception Handling – SEH) je rozšírenie spoločnosti Microsoft pre programovací jazyk C, aby bolo možné elegantne ošetriť určité výnimočné situácie v kóde (napríklad pokus o vykonanie neplatnej inštrukcie). Pre ilustráciu uvažujme nasledovný program v jazyku C:

```
1 BOOL SafeDiv(INT32 dividend, INT32 divisor, INT32 *pResult)
2 {
3   __try
4   {
5     *pResult = dividend / divisor;
6   }
7   __except(GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO
8              ? EXCEPTION_EXECUTE_HANDLER
9              : EXCEPTION_CONTINUE_SEARCH)
```

[2] Java momentálne nevie toto správanie na potomkoch vynútiť.

```
10   {
11     return FALSE;
12   }
13
14   return TRUE;
15 }
```

Odchytenie výnimky zabezpečuje príkaz `__except(filter_expression)`. Pokiaľ má `filter_expression` hodnotu `EXCEPTION_EXECUTE_HANDLER`, tak sa výnimka odchytí. Ak má hodnotu `EXCEPTION_CONTINUE_SEARCH`, tak sa pokračuje v hľadaní vhodného except-bloku. Zaujímavé je uvedomiť si, ako bude prebiehať vykonávanie nasledovného programu:

```
1  #include <exception>
2
3  void sub() {
4    __try {
5      printf("throw ");
6      throw std::exception("");
7    }
8    __finally {
9      printf("finally ");
10   }
11 }
12
13 DWORD filter() {
14   printf("filter ");
15   return EXCEPTION_EXECUTE_HANDLER;
16 }
17
18 void main()
19 {
20   __try {
21     sub();
22   }
23   __except(filter()) {
24     printf("except ");
25   }
26 }
```

Pred akýmkoľvek blokom `finally` sa najprv vyhodnotí `filter_expression` vyššie v zásobníku (aj keď blok `except` spojený s týmto filtrom sa spustí až po skončení zodpovedajúceho bloku `finally`). Predchádzajúci program preto vypíše:

```
throw filter finally except
```

Keďže `filter_expression` sa vyhodnotí pred `finally` príkazom, tak bezpečnostné problémy môžu byť zavedené čímkoľvek, čo zmení stav ochrany, spoliehajúc sa na to, že pred `finally` nie je možné vykonať iný kód. Napríklad:

```
void fun()
  __try {
    Alter_Security_State();//dočasne zvýšime bezpečnostné oprávnenia
    Do_some_work();//vykonáme operáciu potrebujúcu vyššie oprávnenia
  }
  __finally {
    Restore_Security_State(); // vrátime naspäť vyššie oprávnenia
  }
}
```

Vyššie uvedená funkcia `fun()` by sa zdala na prvý pohľad bezpečná. Útočník však môže túto funkciu zavolať z funkcie `attack()` spôsobom ako je uvedené nižšie:

```
void attack()
  DWORD filter() {
    // tento kód je tiež vykonaný s vyššími oprávneniami
    return EXCEPTION_EXECUTE_HANDLER;
  }
  __try { fun() }
  __except() {  }
}
```

Pokiaľ počas vykonania funkcie `fun()` bude vyhodená výnimka, tak útočník získa možnosť vykonať vo funkcii `filter()` akýkoľvek kód s navýšenými oprávneniami.

2.6 Časová závislosť v Dispose

Niektoré jazyky, ako napríklad Java alebo C#, používajú automatické uvoľňovanie dynamicky alokovaných objektov (garbage collection). Vďaka tomu sa programátor nemusí sám explicitne starať o ich uvoľňovanie.

Operačná pamäť je však iba jeden zo zdieľaných prostriedkov, ktorých má systém obmedzené množstvo. Iným môžu byť napríklad popisovače súborov. O uvoľnenie takýchto prostriedkov sa musí stále explicitne postarať programátor. Preto sa takéto prostriedky na .NET platforme nazývajú nespravované (unmanaged resources). Rozhranie `IDisposable` obsahuje metódu `public void Dispose()`, ktorej zavolaním sa práve zabezpečí uvoľnenie nespravovaných zdrojov. Implementácia tejto funkcie býva obvykle podobná ako nasledujúca:

```
public void Dispose()
{
  if (myObj != null)
  {
    Cleanup(myObj);
    myObj = null;
  }
}
```

Ak implementácia `Dispose` nie je synchronizovaná, tak sa môže stať, že metóda `Cleanup` bude zavolaná jedným vláknom a potom aj druhým vláknom ešte predtým, než `myObj` je nastavený prvým vláknom na `null` na riadku 6.

To, či ide o bezpečnostné riziko, závisí od toho, čo sa stane, keď sa opakovane vykoná funkcia `Cleanup`. Vo všeobecnosti opakované uvoľnenie už uvoľneného prostriedku môže viesť k nedefinovanému správaniu aplikácie.

2.7 Práca s celočíselnými premennými

Čísla sú v počítači vždy reprezentované v konečnej pamäti (napríklad v štyroch bajtoch). Preto majú konečnú presnosť. Existuje nejaké najmenšie a najväčšie reprezentovateľné číslo. Pokiaľ by sme chceli najväčšie reprezentovateľné číslo ešte zväčšiť (napríklad pripočítaním jednotky) tak nastane tzv. pretečenie. Pretečenie sa obvykle prejaví zmenou najväčšieho reprezentovateľného čísla na najmenšie reprezentovateľné číslo. Analogický problém nastáva pri zmenšovaní najmenšieho reprezentovateľného čísla. Iným problémom pri práci s číslami je napríklad delenie nulou. Vykonanie takejto operácie môže vrátiť nesprávny/nedefinovaný výsledok alebo vyvolať výnimku.

Pozrime si nasledovný program v jazyku Perl a skúsme odhadnúť, čo vypíše:

```
1  use feature qw(say);
2
3  my $big_int_x = 9223372036854775808; # == 2^63
4  my $big_int_y = $big_int_x - 1;
5
6  my $very_big_int_x = $big_int_x * 2;
7  my $very_big_int_y = $big_int_y * 2;
8
9  say $very_big_int_x; # 1.84467440737096e+19
10 say $very_big_int_y; #  18446744073709551614
11 say $very_big_int_x == $very_big_int_y ? "Same" : "Diff"; # Same
```

Číslo `$big_int_x` nie je zvolené náhodne, ale ide o číslo 2^{63}. Číslo `$big_int_y` je len o 1 menšie. Obe čísla vynásobíme dvomi. Jazyk Perl v snahe predísť pretečeniu pri premennej `$very_big_int_x` skonvertuje pôvodne celé číslo, ktoré už ale prekračuje maximálne reprezentovateľné celé číslo, na číslo v pohyblivej rádovej čiarke. Výsledok je teda stále kladný a v podstate aj celkom presný. Namiesto úplne nesprávnej odpovede v podobe záporného čísla dostaneme číslo 18 446 744 073 709 600 000, ktoré je len o 48 386 väčšie od správneho výsledku. To je chyba len na „zanedbateľnej“ úrovni $-2{,}6229 \times 10^{-13}$%.

Keď na riadku 11 porovnáme `$very_big_int_x` a `$very_big_int_y`, tak prebehne konverzia `$very_big_int_y` na reálne číslo. Keďže pri tejto konverzii opäť príde k drobnej nepresnosti, tak rozdiely v rádu jednotiek sa stratia a porovnávané čísla sa rovnajú, aj keď z matematického hľadiska sú evidentne rôzne.

Program môže byť logicky správny, ale útočník môže zneužiť okrajové správanie aritmetiky celých čísiel v počítači/jazyku a tak zmeniť jeho správanie.

2.8 Práca s premennými s pohyblivou rádovou čiarkou

Reálne čísla sa v počítači najčastejšie reprezentujú ako čísla s pohyblivou rádovou čiarkou[3] v tvare $m \times 2^e$, pričom m je mantisa (v dvojkovej sústave) a e je exponent. Sčítavanie takýchto čísiel potom prebieha tak, že sa najprv zarovnajú mantisy. Teda upravia sa exponenty na rovnaké hodnoty a k tomu sa náležite posunú mantisy (aby sa nezmenila hodnota čísla). Keďže mantisa má predpísaný počet bitov, tak pri týchto posunoch môže z mantisy „vypadnúť“ časť informácie. Samozrejme, snažíme sa zahodiť menej významné bity (radšej malé rády vpravo ako vysoké vľavo).

Preto pri sčítavaní čísiel s veľkými rozdielmi v exponentoch dostávame nepresný výsledok. Napríklad keď sčítame $x = 1\,000\,000\,000$ s $y = 0{,}000\,000\,01$, tak dostaneme presne x, teda $1\,000\,000\,000$. Pripočítaná stomilióntina sa úplne stratila (akoby sme y vôbec nepripočítali). Preto, ak sa dá, treba sčitovať čísla porovnateľných rádov. Inak sa môže stať, že ak aj veľmi veľakrát pripočítame malé číslo y k veľkému x, tak výsledok bude stále iba x.

Iným problémom je skutočnosť, že niektoré čísla nemajú konečnú reprezentáciu. Napríklad $1/3 = 0{,}3333\cdots = 0{,}\overline{3}$. Podobne niektoré „pekné“ čísla z desiatkovej sústavy majú nekonečný „desatinný“ rozvoj v dvojkovej sústave. Napríklad $(0{,}1)_{10} = (0{,}0\overline{0011})_2$. Preto, ak sa číslo s nekonečným „desatinným“ rozvojom v mantise niekde usekne, tak vznikne nepresnosť a číslo reprezentované v počítači bude menšie. Ako príklad si uveďme nasledovný program v jazyku C:

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5    double sum = 0;
6    double delta = 0.1;
7    for(int i = 0; i < 10; i++) sum += delta;
8
9    printf("%f\n", sum);                       // 1.000000
10   printf("%s\n", sum == 1 ? "OK" : "Zle"); // Zle
11   printf("%.17g\n", sum);                    // 0.99999999999999989
12 }
```

Program 10 krát pripočíta k 0 číslo 0,1. Výsledná hodnota premennej `sum` by teda mala byť presne 1. Na riadku 9 vidíme, že výsledok je naozaj 1. Ale na riadku 10 prekvapivo dostaneme odpoveď: Zle (`sum != 1`). Podstatu problému odhalí riadok 11, kde pri použití viac desatinných čísiel vo výpise vidíme, že výsledok je v skutočnosti o čosi málo menší ako 1.

[3] Floating Point Number System – FPNS

Preto, ak napríklad potrebujeme presne pracovať s finančnými čiastkami, môžeme radšej použiť celé čísla a počítať v centoch namiesto v eurách. Prípadne môžeme použiť knižnicu pracujúcu so zlomkami (racionálne čísla + bigint).

2.9 Nebezpečné optimalizácie kompilátora

Zmyslom optimalizácií (vykonávaných kompilátorom pri preklade zdrojového kódu) je dosiahnuť čo najkratší a najrýchlejší výsledný kód pri zachovaní pôvodnej funkčnosti. Niekedy však niečo, čo môže byť na prvý pohľad zbytočné (a preto kompilátorom odstránené), môže byť pre bezpečnosť aplikácie dôležité.

2.9.1 Eliminácia zbytočného kódu

Predstavme si situáciu, keď má program v pamäti uložené dôverné údaje (napríklad šifrovací kľúč alebo heslo). Akonáhle tieto údaje už nepotrebuje, tak ich z pamäte vymaže (prepísaním jej obsahu nulami). Môže ísť napríklad o nasledovný program v jazyku C:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void clear(void) {
5    char pwd[257];
6
7    gets(pwd);
8
9    // použi heslo
10
11   memset(pwd, 0, sizeof(pwd)); // prepíš heslo nulami
12
13   // puts(pwd);
14 }
```

Program najprv načíta heslo do premennej `pwd`. Potom na riadku 9 ho nejako použije. Keď ho už nepotrebuje, tak ho na riadku 11 zavolaním funkcie `memset` prepíše nulami.

Zdrojový kód skompilujeme pomocou optimalizujúceho kompilátora. Kompilátor identifikuje a odstráni prepísanie na riadku 11 ako zbytočné, pretože premenná `pwd` sa už následne nikde nevyužije a preto jej vynulovanie nemá vplyv na výsledok programu. Na obrázku 2.2 je zobrazený vygenerovaný strojový kód, keď sa premenná `pwd` ešte použije na odkomentovanom riadku 13 (vľavo) a keď necháme riadok 13 zakomentovaný (vpravo).

Vidíme, že optimalizujúci kompilátor poctivo odviedol svoju prácu a vygeneroval kratší a rýchlejší kód. Avšak z hľadiska bezpečnosti bola táto optimalizácia nevhodná, keďže heslo ostalo uložené v pamäťových bunkách bez ich prepísania.

```
clear():
  pushq %rbx
  subq  $272, %rsp
  leaq  (%rsp), %rbx
  movq  %rbx, %rdi
  callq gets
  xorl  %esi, %esi
  movl  $257, %edx
  movq  %rbx, %rdi
  callq memset
  movq  %rbx, %rdi
  callq puts
  addq  $272, %rsp
  popq  %rbx
  retq
```

```
clear():

  subq  $264, %rsp
  leaq  (%rsp), %rdi

  callq gets









  addq  $264, %rsp

  retq
```

Obr. 2.2: Eliminácia zbytočného kódu

2.9.2 Pretečenie smerníkovej aritmetiky

Definícia jazyka C hovorí, že správanie programu je pri pretečení aritmetiky na smerníkoch nedefinované. Predstavme si nasledovný program, ktorý sa snaží zistiť, či smerník `buf` nepretečie pri práci s `len` prvkami:

```
#include <stdio.h>

void wrap(unsigned long len)
{
  char *buf;

  if (buf + len < buf)
  {
    puts("Pretečenie");
  }
}
```

Ako už vieme z predošlých častí, ak súčet pretečie, tak výsledok bude menší ako `buf` a program by mal vypísať „Pretečenie“. Ak ale k pretečeniu príde, tak táto kontrola vlastne zahŕňa výpočet neplatného smerníka (pri ktorom prišlo k pretečeniu) a preto je nasledovné správanie programu nedefinované. Z tohto dôvodu sa niektoré optimalizujúce kompilátory môžu rozhodnúť pre výrazne zjednodušujúcu transformáciu, a to, že podmienka bude vždy nesplnená. Tým sa nechcene z programu úplne odstráni kontrola hraníc. Vidíme to aj na výslednom preklade vyššie uvedeného kódu:

```
wrap(unsigned long):    # @wrap(unsigned long)
        retq
```

Aj keď podmienka $buf + len < buf$ môže byť splnená (pri aritmetickom pretečení), tak kompilátor volanie puts pri optimalizácii úplne odstráni.

2.9.3 Nedefinované správanie

Problém z predošlej podčasti bol len špeciálnym prípadom všeobecnejšieho problému, ktorým je nedefinované správanie. Iným príkladom nedefinovaného správania v jazyku C je dereferencovanie `NULL` pointra.

V špecifikácii programovacích jazykov sa s nedefinovaným správaním môžeme stretnúť relatívne často. Dôvodom nie je lenivosť návrhárov jazyka, ale ich snaha pomôcť tvorcom optimalizujúcich kompilátorov generovať optimálny kód. Túto problematiku si priblížime na nasledovnom programe v jazyku C:

```
1  #include <cstdlib>
2
3  typedef int (*Function)();
4
5  static Function Do;
6
7  static int EraseAll() {
8    return system("rm -rf /");
9  }
10
11 void NeverCalled() {
12   Do = EraseAll;
13 }
14
15 int main() {
16   return Do(); // zavolanie neinicializovaného (NULL)
17      // smerníka na funkciu -> nedefinované správanie
18 }
```

Vidíme, že premenná `Do` je typu pointer na funkciu bez parametrov, ktorá vracia hodnotu typu `int`. Program na riadku 16 zavolá funkciu, na ktorú ukazuje premenná `Do`. Preto kompilátor spraví analýzu kódu, s cieľom zistiť, aké hodnoty môže premenná `Do` na riadku 16 nadobúdať. Vidí, že na riadku 5 je premenná deklarovaná, ale nie je inicializovaná. Takže jedna potenciálna hodnota premennej `Do` je `NULL`. Jediné miesto v programe, kde sa do premennej `Do` priraďuje nejaká hodnota je riadok 12. Preto usúdi, že $\texttt{Do} \in \{\texttt{NULL}, \texttt{EraseAll}\}$. My vieme, že `Do` bude `NULL`. Ale predpokladajme, že kompilátor nevie hodnotu `EraseAll` s istotou vylúčiť. Preto sa radšej konzervatívne[4] rozhodne, že ju pripustí.

Teraz príde na pomoc nedefinované správanie. Keďže kompilátor vie, že ak je $\texttt{Do} = \texttt{NULL}$, tak môže na riadku 16 spraviť čokoľvek, rozhodne sa na tomto riadku z množiny možných hodnôt pre `Do` vynechať `NULL`. Ostane mu teda jediná možná hodnota a to `EraseAll`. V ta-

[4] V najhoršom potom nebude môcť spraviť optimalizáciu, ale vytvorený kód bude určite správny.

komto prípade môže nepriame volanie funkcie cez smerník zoptimalizovať na priame zavolanie jedinej možnej funkcie `EraseAll`. Výsledný kód vidíme nižšie:

```
1 NeverCalled():                        # @NeverCalled()
2         ret
3 main:                                 # @main
4         mov     edi, offset .L.str
5         jmp     system                # TAILCALL
6 .L.str:
7         .asciz  "rm -rf /"
```

Vygenerovaný program hneď po svojom spustení vymaže všetky súbory na disku. Názorne sme si teda ukázali, že nedefinované správanie nemusí skončiť len pádom aplikácie, ale naozaj môže viesť k vykonaniu takmer čohokoľvek.

Kapitola 3

Záver

Naším cieľom bolo poukázať na dôležitosť a široký záber problematiky bezpečného programovania. Venovali sme sa problematike dočasných súborov, kde sme ukázali odporúčaný spôsob ich používania. Ďalej sme si na príklade ukázali zneužitie pretečenia vyrovnávacej pamäte („buffer overflow“). Na konkrétnom príklade sme si vysvetlili aj fungovanie „SQL injection“, ako špeciálneho prípadu útoku vložením kódu („code injection“). Potom sme spomenuli špecifiká programovacích jazykov, ako napríklad modifikátor `final` v jazyku Java a prečo je napríklad trieda `String` v Jave `final`.

Ďalej sme sa venovali štruktúrovaným výnimkám na platforme Windows a ukázali sme, ako môže útočník pomocou filtra vykonať kód ešte pred vykonaním bloku `__finally`. Na príklade jazyka C# sme si spomenuli problémy s časovou závislosťou pri uvoľňovaní zdrojov. Uviedli sme aj niekoľko problémov pri práci s celými a reálnymi číslami. Na záver učebného textu sme podrobnejšie popísali bezpečnostné implikácie niektorých optimalizácií moderných kompilátorov.

Bibliografia

[1] D. Deogun, D. B. Johnsson a D. Sawano. *Secure by design*. Shelter Island: Manning Publications, 2019. ISBN: 9781617294358 (citované na strane 1).

[2] M. Howard a D. LeBlanc. *Writing secure code*. 2. vydanie. Redmond, Wash: Microsoft Press, 2003. ISBN: 9780735617223 (citované na strane 1).

[3] B. Chess a J. West. *Secure programming with static analysis*. Addison-Wesley software security series. Upper Saddle River, NJ: Addison-Wesley, 2007. ISBN: 9780321424778 (citované na strane 1).

[4] F. Long. *Java coding guidelines: 75 recommendations for reliable and secure programs*. The SEI series in software engineering. Upper Saddle River, NJ: Addison-Wesley, 2014. ISBN: 9780321933157 (citované na strane 1).

[5] M. S. Merkow a L. Raghavan. *Secure and resilient software development*. Boca Raton, FL: CRC Press, 2010. ISBN: 9781439826966 (citované na strane 1).

[6] *Secure Coding Guidelines for Java SE*. Technická správa. Oracle, sept. 2020. URL: https://www.oracle.com/java/technologies/javase/seccodeguide.html (citované na strane 1).

[7] *SEI CERT C Coding Standard – Rules for Developing Safe, Reliable, and Secure Systems*. Technická správa. Pittsburgh, PA: Carnegie Mellon University – Software Engineering Institute, jún 2016. URL: https://resources.sei.cmu.edu/forms/secure-coding-form.cfm (citované na strane 1).

[8] *SEI CERT Coding Standards*. Technická správa. Pittsburgh, PA: Carnegie Mellon University – Software Engineering Institute, 2020. URL: https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards (citované na strane 1).

[9] J. Viega a M. Messier. *Secure programming cookbook for C and C++*. 1. vydanie. Cambridge: O'Reilly, 2003. ISBN: 9780596003944 (citované na strane 1).

[10] D. A. Wheeler. *Secure Programming for Linux and Unix HOWTO*. Mar. 2003. URL: https://tldp.org/HOWTO/pdf/Secure-Programs-HOWTO.pdf (citované na strane 1).