Bezpečné programovanie 2

RNDr. Richard Ostertág, PhD. (ostertag@dcs.fmph.uniba.sk)

Katedra informatiky
Univerzita Komenského, Bratislava

Dočasné súbory 1/5

- Aplikácie často potrebujú používať dočasné súbory.
- Často sa ukladajú do adresárov, ktoré sú prístupné všetkým aj na zápis.
    - /tmp
    - /var/tmp
    - C:\Windows\TEMP
    - C:\Users\Name\AppData\Local\Temp
    - ...
- Dočasné súbory sa môžu zmazať:
    - hneď, keď ich aplikácia nepotrebuje
    - keď aplikácia skončí
    - počas štartu/vypínania operačného systémy
    - raz za deň
    - ...

Dočasné súbory 2/5

- Dočasné súbory musia mať nepredpovedateľné meno
  - inak privilegovaný program môže prepísať chránené súbory
    - útočník vytvorí v /tmp symbolickú linku s predpovedaným menom na chránený súbor
  - inak neprivilegovaný program môže prepísať používateľove súbory
    - útočník vytvorí v /tmp symbolickú linku s predpovedaným menom na používateľov súbor
- `tmpnam()` alebo `mktemp()` vytvoria taký názov súboru:

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

Dočasné súbory 3/5

- Time of Check to Time of Use (TOCTOU)
  - Medzi získaním mena a vytvorením súboru prejde istý čas.
    - Pre vyriešenie tejto časovej závyslosti je potrebná podpora OS.
  - V tomto čase môže niekto získané meno obsadiť.

- Predísť časovej závislosti je možné použitím funkcií vracajúcich priamo súbor:
  - `tmpfile()`
  - `mkstemp()`

- `tmpfile()` otvorí jedinečný dočasný súbor v binárom read/write (w+b) móde.
  - Súbor bude automaticky vymazaný, keď bude zatvorený alebo program ukončený.
  - `FILE *tempfile = tmpfile(void);`

Dočasné súbory 4/5

- Dočasné súbory musia byť otvorené s exkluzívnym prístupom a vhodnými prístupovými právami.
- Program by mal odstrániť svoje dočasné súbory skôr než skončí.
  - šetrí sa diskový priestor
  - znižuje sa šanca, že v budúcnosti nastane kolízia
- Opustené dočasné súbory nie sú zriedkavé $\Rightarrow$ rôzne nástroje na čistenie adresárov s dočasnými súbormi
  - manuálne administrátorom
  - cron daemon odstráni niekoľko dní staré dočasné súbory
  - vymažú sa pri štarte/vypínaní systému
- Tieto nástroje sú tiež náchylné na útoky
  - nahradením dočasného súboru symbolickou linkou na iný súbor
  - priame vytvorenie symbolickej linky na iný súbor

Dočasné súbory 5/5

```c
char sfn[15] = "/tmp/ed.XXXXXX";
FILE *sfp; int fd = -1;
if ((fd = mkstemp(sfn)) == -1 ||
    (sfp = fdopen(fd, "w+")) == NULL) {
  if (fd != -1) {
    unlink(sfn); close(fd);
  }
  /* handle error condition */
}
unlink(sfn); /* unlink immediately */
/* use temporary file */
close(fd);
```

Pokiaľ existuje proces, ktorý má súbor otvorený, unlink() odstráni súbor z adresára, ale fyzicky ho vymaže až keď ho všetky procesy zavrú.

Java – modifikátor final

- Od `final` triedy nie je možné dediť.
- `final` metóda nemôže byť predefinovaná v potomkoch.
- Do `final` premennej je možné priradiť iba raz.
- Keď trieda, ani metóda nie je `final`, tak útočník môže vytvoriť potomka s predefinovanou metódou.
    - To môže viesť k neočakávanému správaniu.
    - Napríklad rodič môže byť immutable ale potomok už nie.
    - Pokiaľ sa pri viacvláknových aplikáciách zdieľajú immutable objekty, tak zmena tejto vlastnosti u potomka je vážny problém.
- Strata rozšíriteľnosti vs. bezpečnosť.
    - Napríklad pri `String` nechceme aby niekto predefinoval:
        - že je `String` immutable
        - ako pracuje `compareTo`
        - preto je trieda `String` v Jave `final`

Buffer overflow – program

```c
// save to attack.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);
}
```

Buffer overflow – zistenie umiestnenia premennej password

```
gcc -no-pie attack.c
objdump -x a.out | grep password
0000000000601040 g     O .data  0000000000000008              password
```

umiestnenie globálnej premennej password v dátovom segmente
(je to iba ukazovateľ na reťazec)

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

Contents of section .data:

```
601030 00000000 00000000 00000000 00000000  ................
601040 94064000 00000000                    ..@.....
```

takže tajný reťazec je na adrese: 0x400694

Buffer overflow – vzdialené zneužitie

```
perl -e 'print "a"x104; print "\x94\x06\x40"; print "\x00"x5' | ./a.out

Enter user name: ... processing user name ...
SuperSecretPassword123
```

Structured exception handling – SEH 1/3

Štruktúrované spracovanie výnimiek je rozšírenie Microsoftu pre C, aby bolo možné elegantne ošetriť určité výnimočné situácie v kóde, ako napríklad neplatná inštrukcia.

```
1  #include <exception>
2  void sub() {
3    __try {
4      printf("throw\n");
5      throw std::exception("");
6    }
7    __finally { printf("finally\n"); }
8  }
9  bool filter() {printf("filter\n"); return true;}
10 void main()
11 {
12   __try { sub(); }
13   __except (filter()) { printf("catch\n"); }
14 }
```

Structured exception handling – SEH 2/3

Vo Visual C++ a Visual Basic sa `filter`-výraz vyššie v zásobníku spúšťa pred akýmkoľvek `finally` príkazom. Blok **`catch`** spojený s týmto filtrom sa spustí až po `finally` príkaze. Predchádzajúci kód vypíše nasledovné:

```
throw
filter
finally
catch
```

Structured exception handling – SEH 3/3

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

```
1  try {
2      Alter_Security_State();
3      // Toto môže byť čokoľvek čo môže byť
4      // zneužité ak sa nepriateľský kód spustí
5      // pred obnovením pôvodného stavu.
6      Do_some_work();
7  }
8  finally {
9      Restore_Security_State();
10     // Vráti naspäť vyššie urobenú zmenu stavu.
11 }
```

.NET Framework – časová zavislosť v Dispose

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`.

To, či ide o bezpečnostné riziko, závisí od toho, čo sa stane, keď sa opakovane spustí kód `Cleanup`.

```
1 void Dispose()
2 {
3     if ( myObj != null )
4     {
5         Cleanup(myObj);
6         myObj = null;
7     }
8 }
```

Práca s celočíselnými premennými – integer

- delenie nulou
- pretečenie (do záporných čísiel, na nulu, iné)

```
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
```

Práca s reálnymi premennými – double

- sčítavanie čísiel s veľkými rozdieľmi v exponentoch je nepresné:

$$1\,000\,000\,000 + 0{,}000\,000\,01 = 1\,000\,000\,000$$

- niektoré pekné čísla v desiatkovej sústave majú nekonečný desatinný rozvoj v dvojkovej sústave, napríklad: $(0{,}1)_{10} = (0{,}000110011001100110011\ldots)_2$.

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  double sum = 0;
  double delta = 0.1;
  for(int i = 0; i < 10; i++) sum += delta;

  printf("%f\n", sum);                          // 1.000000
  printf("%s\n", sum == 1 ? "OK" : "Zle"); // Zle
  printf("%.17g\n", sum);  // 0.99999999999999989
}
```

Vkladanie riadiacich dát – code injection

- každý jazyk má špecifickú syntax a špecifické riadiace znaky
- pri implemtnácii web aplikácie sa často používajú rôzne jazyky (napr.: SQL, HTML, JavaScript, XML, HTTP, …)
- ak sa neošetrené dáta od klienta vložia do konštrukcie v niektorom z týchto jazykov, môžu nadobudnúť neočakávaný význam
  - SQL injection
  - XPath injection
  - HTML injection (markup injection)
    CSRF, XSRF: cross-site request forgery
  - JavaScript injection
    XSS: cross-site scripting

SQL injection

- select * from Pouzivatelia where Meno = ’Janko Hrasko’;
- vstup: ’ or ’’ = ’

  select * from Pouzivatelia where Meno = ’’ or ’’ = ’’;
- vstup: ’ and 1 = 0 union all select * from Tabulka; --

  select * from Pouzivatelia where Meno = ’’ and 1 = 0 union all select * from Tabulka; --’;
- niekedy aplikácia vráti iba časť výsledku alebo vracia informáciu iba nepriamo
  - chybová hláška
  - dĺžka spracovania požiadavky

SQL injection – skutočný príklad :-)

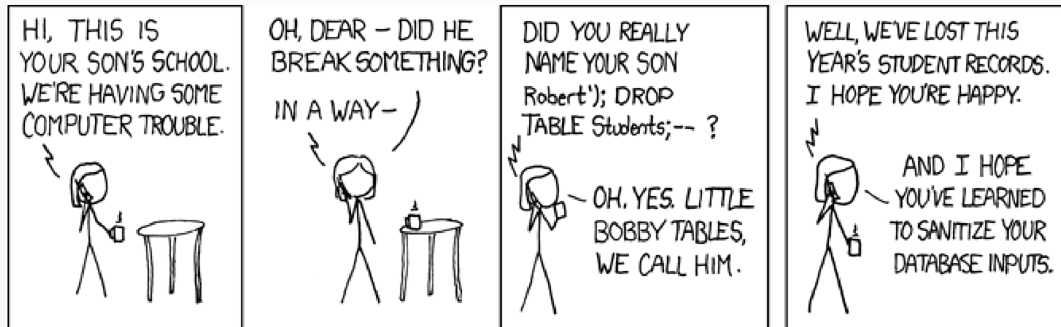
Gijs in 't Veld
@gintveld

Sledovať

Great to see that my name still causes SQL errors and that errors thrown are so hacker friendly. ;-) #integrate2016

Invalid query: You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near 't Veld', NOW()), ('cc89fdd01ea0', 'User-Profile', ':=', 'free750', '', '', NOW(' at line 1 Whole query: INSERT INTO newusers (username, attribute, op, value, callingstationid, displayname, created_at) VALUES ('cc89fdd01ea0', 'Cleartext-Password', ':=', '70358542', 'cc-89-fd-d0-1e-a0', 'Gijs in 't Veld', NOW()),('cc89fdd01ea0', 'User-Profile', ':=', 'free750', '', '', NOW())

SQL injection – menej skutočný príklad :-)[1]



[1] Zdroj: https://xkcd.com/327/ (Exploits of a Mom).

Nebezpečné optimalizácie kompilátora – memset

- V pamäti sú uložené tajné údaje.
- Tajné údaje sa z pamäte vymažú prepísaním jej obsahu.
- Zdrojový kód je skompilovaný pomocou optimalizujúceho kompilátora.
- Kompilátor identifikuje a odstráni prepísanie ako zbytočné, pretože pamäť sa už následne nevyužije.

Nebezpečné optimalizácie kompilátora – memset príklad

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

void clear(void) {
  char pwd[257];
  gets(pwd);
  memset(pwd, 0, sizeof(pwd));
  puts(pwd); // tento riadok neskôr zakomentujeme
}
```

Nebezpečné optimalizácie kompilátora – s puts

```
1  clear():                                # @clear()
2          pushq   %rbx
3          subq    $272, %rsp                  # imm = 0x110
4          leaq    (%rsp), %rbx
5          movq    %rbx, %rdi
6          callq   gets
7          xorl    %esi, %esi
8          movl    $257, %edx                  # imm = 0x101
9          movq    %rbx, %rdi
10         callq   memset
11         movq    %rbx, %rdi
12         callq   puts
13         addq    $272, %rsp                  # imm = 0x110
14         popq    %rbx
15         retq
```

Nebezpečné optimalizácie kompilátora – bez puts

```
 1  clear():                                         # @clear()
 2
 3          subq    $264, %rsp                         # imm = 0x108
 4          leaq    (%rsp), %rdi
 5
 6          callq   gets
 7
 8
 9
10
11
12
13          addq    $264, %rsp                         # imm = 0x108
14
15          retq
```

Nebezpečné optimalizácie kompilátora – smerníková aritmetika

- Správanie pri pretečení aritmetiky na smerníkoch je nedefinované.
- Kontrola hraníc poľa by sa mohla pri optimalizácii nechcene odstrániť.
- Ak kontrola hraníc poľa zahŕňa výpočet neplatného smerníka s následným testom, či je smerník mimo hraníc, tak niektoré kompilátory pri optimalizácii kódu túto kontrolu z neho odstránia.

Nebezpečné optimalizácie kompilátora – smerníková aritmetika

```c
#include <stdio.h>
void wrap(unsigned long len) {
  char *buf;
  if (buf + len < buf)
    puts("OK");
}
```

vyššie uvedený program sa preloží na:

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

Aj keď podmienka $buf + len < buf$ môže byť splnená (kontrola aritmetického pretečenia), tak kompilátor volanie puts pri optimalizácii odstráni.

Nebezpečné optimalizácie kompilátora – nedefinované správanie

```cpp
#include <cstdlib>
typedef int (*Function)();
static Function Do;

static int EraseAll() {
  return system("rm -rf /");
}
void NeverCalled() {
  Do = EraseAll;
}
int main() {
  return Do(); // zavolanie neinicializovaného (null)
}              // smerníka na funkciu -> nedef. správanie
```

Nebezpečné optimalizácie kompilátora – nedefinované správanie

```
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 /"
```

Kompilátor po zoptimalizovaní kódu vo funkcii main zmaže všetky súbory.