



Agentúra

Ministerstva školstva, vedy, výskumu a športu SR
pre štrukturálne fondy EÚ



Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Príprava štúdia matematiky a informatiky na FMFI UK v anglickom jazyku

ITMS: 26140230008

dopytovo – orientovaný projekt



Moderné vzdelávanie pre vedomostnú spoločnosť/Projekt je spolufinancovaný zo zdrojov EÚ



Resources

- ▶ CERT Secure Coding Standards (for Java, C, C++)
<https://www.securecoding.cert.org/>
- ▶ Secure Coding Guidelines for the Java Programming Language
<http://www.oracle.com/technetwork/java/seccodeguide-139067.html>
- ▶ Secure Programming for Linux and Unix HOWTO
<http://www.dwheeler.com/secure-programs/>
- ▶ M.Sc. Pascal Meunier, Ph.D.: Overview of Secure Programming

Public vulnerability databases and resources

- ▶ MITRE's CVE (<http://cve.mitre.org/>)
 - ▶ Common Vulnerabilities and Exposures
 - ▶ Common Vulnerabilities Enumeration
- ▶ MITRE's CWE (<http://cwe.mitre.org/>)
 - ▶ Common Weakness Enumeration
 - ▶ Comprehensive CWE Dictionary
<http://cwe.mitre.org/data/slices/2000.html>
 - ▶ Top 25 Most Dangerous Software Errors
<http://cwe.mitre.org/top25/>
- ▶ NIST's ICAT (<http://icat.nist.gov/>)
 - ▶ based on the CVE
 - ▶ completes vendor and product information
 - ▶ adds a classification of vulnerabilities

SD3 – Secure by Design, by Default, in Deployment

- ▶ the system should be designed with security on mind from the beginning
- ▶ the developer should know what options are dangerous
- ▶ all dangerous options should have appropriate default values
- ▶ customer doesn't know the system any better so the installation and configuration program should provide reasonable defaults
 - ▶ exceptions should provide warnings

Code analysis

- ▶ static (i.e. before the code execution)
 - ▶ peer review of design and code (e.g. code review)
 - ▶ applications for coding style verification
 - ▶ applications for static program analysis
 - ▶ unused variables
 - ▶ uninitialized variable
 - ▶ these problems are hard \Rightarrow only conservative approximation
- ▶ dynamic (i.e. checks during runtime to see whether the code meets the model)
 - ▶ checking of invariants
 - ▶ pre-conditions and post-conditions
 - ▶ all allocated memory is released
 - ▶ assert

Filename extensions of executable files (Windows NT)

- ▶ After entering command without extension, system gradually tests extensions from the PATHEXT environment variable.
- ▶ Default value is ".COM;.EXE;.BAT;.CMD".
- ▶ Attacker can change executed application by changing the value of the PATHEXT environment variable.
- ▶ Similar problems are caused by the PATH environment variable.
 - ▶ Determines the order of directories in which the system is looking for program.
 - ▶ relevant also for Linux

White list vs. black list

- ▶ security should not be based on enumeration of each dangerous thing (black list)
 - ▶ it's easy to miss somethings
- ▶ instead, security should be based on denying everything by default, unless something is explicitly enumerated as safe (white list)
- ▶ example of incorrect fix approach:
 - ▶ try to block a specific exploitation path by using black list
 - ▶ the attacker will likely find another path which bypasses the black list

Format string vulnerabilities – C

arises by insertion of untrusted data into a format string

- ▶ What is the format string?
 - ▶ `printf ("Name: %s (age: %11d)", person, age);`
Name: Einstein (age: 133)
- ▶ especially dangerous is "%n"
 - ▶ "Nothing printed. The argument must be a pointer to a signed int, where the number of characters written so far is stored."
- ▶ not knowing that the function interprets the text as the format string
 - `snprintf(str, sizeof(str), "Wrong password (user %s)", username);`
 - `syslog(LOG_WARNING, str);`
 - ▶ `syslog()` uses its second argument as a format string
 - ▶ `username = "einstein%s%s%s%s"` likely to cause application crash
- ▶ wrong way of string printing: `fprintf(log, logmessage);`
 - ▶ correct way: `fprintf(log, "%s", logmessage);`

Format string vulnerabilities – PHP

- ▶ PHP does not support "%n"
- ▶ let format3.php have the following content:

```
#!/usr/bin/php
<?php
printf("%s", "Hello 1!\n");
printf("%s%s", "Hello 2!\n");
printf("%s", "Hello 3!\n");
?>
```

- ▶ format3.php outputs:

Hello 1!

PHP Warning: printf(): Too few arguments in format3.php on line 4
Hello 3!

- ▶ program continues, outputting only the empty string
- ▶ can be used to suppress log messages

Format string vulnerabilities – Python

- ▶ Python does not support "%n", does not have printf, but does contain the % (format) command.

- ▶ let format4.py have the following content:

```
#!/usr/bin/python
userdata = {"user": "admin", "pass": "usr123"}
passwd = raw_input("Password: ")
if (passwd != userdata["pass"]):
    print ("Wrong password: " + passwd) % userdata
else:
    print "Welcome %(user)s!" % userdata
```

- ▶ after executing format4.py, the attacker can enter the magic password:
Password: %(pass)s
Wrong password: usr123
- ▶ inconsistency in number of % and arguments leads to an exception
 - ▶ can be used to suppress log messages (if improperly treated)
 - ▶ can lead to DoS attack (if not catched)

Resource exhaustion 1

- ▶ shared resources are exposed to attacks
 - ▶ operating memory
 - ▶ disk space
 - ▶ network bandwidth
 - ▶ CPU
 - ▶ entropy (for random number generation)
 - ▶ process table
 - ▶ file descriptors
 - ▶ database and other servers
 - ▶ analysts
 - ▶ ...
- ▶ may occur if there is only:
 - ▶ finite number of resources
 - ▶ finite amount (e.g. of memory)
 - ▶ finite performance (e.g. of CPU)

Resource exhaustion 2

- ▶ generous protocols and algorithms
 - ▶ unauthenticated usage of computer resources, unwise operational order
 - ▶ performing a series of complex operations before checking the request's validity
 - ▶ e.g. server generates keys right after the connection is established, prior the user is authenticated
 - ▶ amplification (via broadcast, subscriptions, . . .), asymmetric attacks
 - ▶ cost for attacker is much smaller than for defender
 - ▶ ICMP ping to broadcast address with a spoofed source IP address of victim (smurf attack)
- ▶ coding errors turned into vulnerabilities
 - ▶ memory leaks
- ▶ design errors
 - ▶ absence of access control
 - ▶ absence of restrictions on resource utilization
 - ▶ . . .

Resource exhaustion 3

- ▶ algorithmic complexity attacks
 - ▶ exploit worst-case scenario of algorithms
 - ▶ quicksort: $O(n \log n) \rightarrow O(n^2)$
 - ▶ hash tables: $O(n) \rightarrow O(n^2)$
 - ▶ regular expressions: $O(n) \rightarrow O(2^n)$
 - ▶ to fix use algorithms that are not vulnerable
 - ▶ *universal hash algorithms* designed to avoid the vulnerability
- ▶ statefull protocols are necessarily more vulnerable to DoS attacks
- ▶ to fix, convert them into stateless protocols
 - ▶ idea: encrypt the state data, and return it to client
 - ▶ no memory usage
 - ▶ increased CPU and bandwidth usage trade-off

ReDoS – regular expression denial of service attack

- ▶ after translation of regular expression to NFA with m states, we can proceed as follow:
 - ▶ NFA \rightarrow DFA [conversion $O(2^m)$ usually $O(m)$, searching $O(n)$]
 - ▶ backtracking path in NFA [searching $O(2^n)$ usually $O(n)$]
 - ▶ backtracking all paths in NFA in parallel [$O(m^2n)$]
 - ▶ lazy conversion to DFA during searching [$O(m^2n)$]
- ▶ typical evil regular expression is " $^((a+)^+)^+$$ "
- ▶ the attacker can apply ReDoS attack, if he/she can:
 - ▶ enter input to exploitable RE (e.g. "aaaaaaaaaa!")
 - ▶ OWASP Validation Regex Repository:
Java Classname: `^(([a-z])+.)+([A-Z])([a-z])+$`
 - ▶ `cregex = re.compile(r"^(([a-z])+.)+([A-Z])([a-z])+$")`
 - ▶ `cregex = re.compile(r"^((([a-z])+.)+([A-Z])([a-z])+$")`
 - ▶ `match = cregex.match("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!")`;
 - ▶ enter subexpression into RE (with appropriate RE input)

Faulty memory management

- ▶ memory leaks
- ▶ double free
- ▶ use of freed memory
- ▶ freeing wrong memory
- ▶ memory access to an invalid address
- ▶ information leakage
 - ▶ overwrite sensitive memory to prevent leakage
 - ▶ prevent passwords and keys from being saved to disk
 - ▶ virtual memory, swap space – use memory locking
 - ▶ crash dumps (core files) – disable crash dumps

Temporary Files 1

- ▶ applications frequently need to use temporary files
- ▶ often stored in directories accessible to all even for writing
 - ▶ /tmp
 - ▶ /var/tmp
 - ▶ C:\Windows\TEMP
 - ▶ C:\Users\Name\AppData\Local\Temp
 - ▶ ...
- ▶ temporary files may be deleted
 - ▶ as soon as the application does not need them
 - ▶ when the application terminates
 - ▶ during startup or shutdown of operating system
 - ▶ once a day
 - ▶ ...

Temporary Files 2

- ▶ temporary files must have an unpredictable name
 - ▶ otherwise privileged program can overwrite protected files
 - ▶ attacker can create in the /tmp directory symbolic link with predictable name to the protected file
 - ▶ otherwise unprivileged program can overwrite user's files
 - ▶ attacker can create in the /tmp directory symbolic link with predictable name to the user file
- ▶ `tmpnam()` or `mktemp()` create a such a file name:

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

Temporary Files 3

- ▶ Time of Check to Time of Use (TOCTOU)
 - ▶ some time passes between obtaining the file name and the file creation
 - ▶ to overcome this race condition OS support is needed
 - ▶ in this time somebody can create symbolic link with the same file name
- ▶ to avoid this race conditions, functions directly returning open file descriptor should be used:
 - ▶ `tmpfile()`
 - ▶ `mkstemp()`
- ▶ `tmpfile()` opens a unique temp. file in binary read/write (`w+b`) mode
 - ▶ the file will be automatically deleted when it is closed or the program terminates
 - ▶ `FILE *tempfile = tmpfile(void);`

Temporary Files 4

- ▶ temporary files must be opened with exclusive access and appropriate access rights
- ▶ program should remove its temporary files before termination
 - ▶ saves the disk space
 - ▶ reduces the chance that a collision will occur in the future
- ▶ abandoned temporary files are not rare ⇒ variety of tools to clean temporary directories
 - ▶ manually by the administrator
 - ▶ cron daemon deleting a few days old temporary files
 - ▶ cleaning at system startup
- ▶ these tools are also prone to attacks
 - ▶ by replacing the temporary file with symbolic link to another file
 - ▶ by direct creation of symbolic link to another file

Temporary Files 5

```
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);
```

If there is a process that has the file open, `unlink()` only removes the file from the directory, but the file is physically deleted later, when it is closed by all processes.

Java – final modifier

- ▶ final class can not be extended
- ▶ final method can not be overridden in subclasses
- ▶ final variable can be assigned only once
- ▶ if class neither method is not final, then attacker can create subclass with overridden method
- ▶ this can leads to unexpected behavior
- ▶ extensibility vs. security