Systems and Software Security Session – Writing Secure Code Not Being a Security Specialist

Gerard Frankowski

Poznań Supercomputing and Networking Center

www.eu-egee.org

EGEE and gLite are registered trademarks

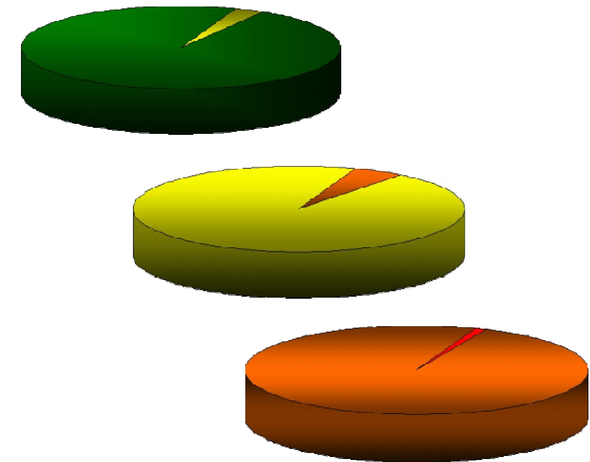
Agenda

- **Secure coding – introduction**
  - Why the code is insecure?
  - Why the developers should care?
- **Input data sanitization**
- **Coding errors: examples and avoiding**
  - Standalone applications
  - Web applications
- **Other general issues**
  - Handling sensitive data in memory
  - Dangerous functions and APIs
  - Coding conventions and comments
- **Where to find more information?**
- **Questions**

Why the developer should care?

Why the code is insecure?

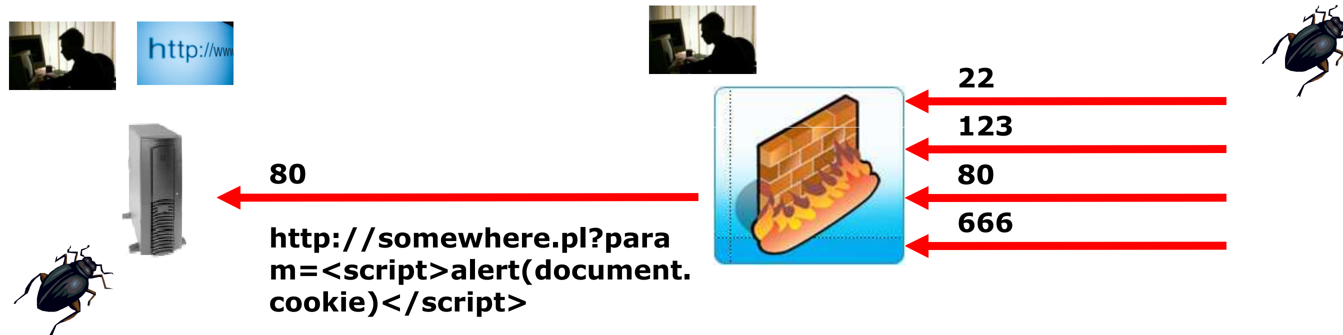
- **We are not robots, we do mistakes**
  - Let the software be 30 000 KLOC (30 millions lines) long
    - Windows 2000 was of that size
  - According to the Carnegie Mellon University's CyLab, 1 KLOC (1000 lines of code) contains up to 30 software bugs
  - Let's make some further assumptions:
    - 20 software bugs (of all kinds) in 1 KLOC
    - only 5% of them are security-related
    - only 1% of the latter give system access



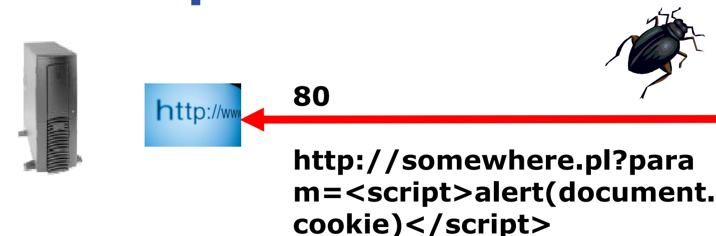
  - 30 000 000 * 0.02 * 0.05 * 0.01 = 300
  - The attacker needs to find only 1 out of those 300…

Why secure code does matter?

- **It is the administrator who should take care on the system security, isn't it?**
  - Appropriate server configuration
  - Firewall policies…



- **The response must be „defence in-depth"**
  - We defend on every level



Are the software bugs expensive?

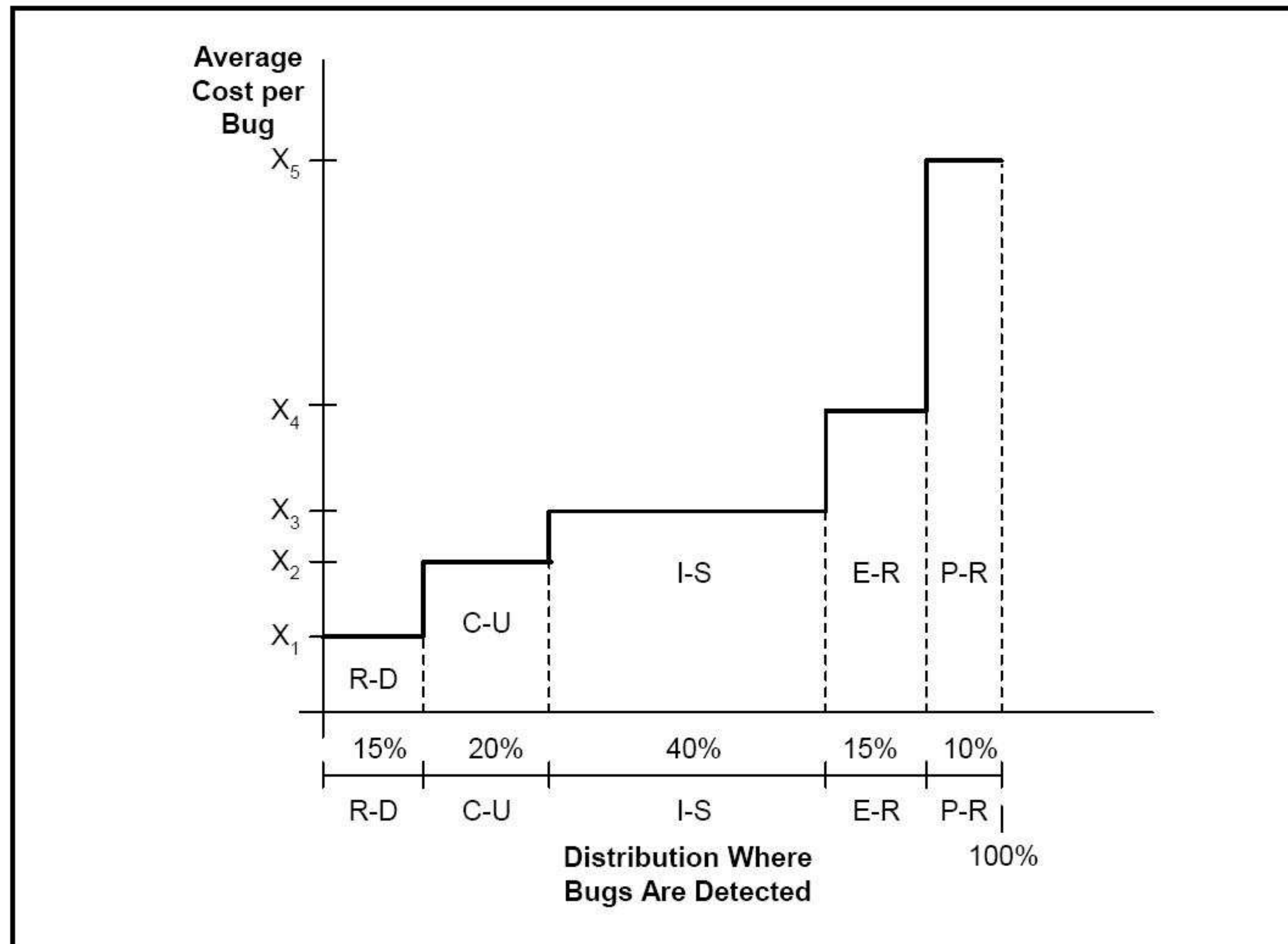
- **NIST Report "The Economic Impacts of Inadequate Infrastructure for Software Testing" (2002)**
  - http://www.nist.gov/director/prog-ofc/report02-3.pdf
  - This is more general, not only security bugs
    - But they are bugs as well…

Table 7-5. Hours to Fix Bug based on Introduction Point

	Stage Found				
Stage Introduced	**Requirements**	**Coding/Unit Testing**	**Integration**	**Beta Testing**	**Post-product Release**
Requirements	1.2	8.8	14.8	15.0	18.7
Coding/unit testing	NA	3.2	9.7	12.2	14.8
Integration	NA	NA	6.7	12.0	17.3

NA = Not applicable because cannot find a bug before it is introduced

Again the NIST report…



Legend:
- R-D: Requirements Gathering and Analysis/Architectural Design
- C-U: Coding/Unit Test
- I-S: Integration and Component/RAISE System Test
- E-R: Early Customer Feedback/Beta Test Programs
- P-R: Post-product Release

Should the developer be a security specialist then?

- **Obviously not!**
  - Everyone has got his or her own job
  - We do not expect the developers to learn about network attacks or exploiting
- **But we think we can expect the developers…**
  - To know the basics of secure coding (including simple examples of attacks for better understanding)
  - To apply secure coding practices in their favourite programming language (or the one they have to work with…)
  - To create well-commented and easy-to-understand code
  - To apply simple tools detecting the most obvious security flaws
- **…and we would like to help by sharing our knowledge and experience!**
  - Last but not least, a „secure coder" will be more competitive ;)

Several spectacular software bugs

- **Therac 25 (1980's)**
  - Radiation overdosing, at least 5 deaths
- **Research on Mars**
  - Mars Global Surveyor (2006)
  - Mars Exploitation Rover (2004)
  - Mars Climate Orbiter (1999) – price: 125 mln USD
- **Northeast Blackout (2003)**
  - Price: 7-10 mld USD
- **Ariane 5 Explosion (1997)**
  - Price: 500 mln USD
- **More information:**
  - http://computingcases.org/case_materials/therac/therac
  - http://www.cse.lehigh.edu/~gtan/bug/softwarebug.html

But if the software will never be perfect…

- **Why we should care?**
- **Security economy!**
  - The economic factors become more meaningful for everyone, including hackers
  - Your system is in danger when

Attack cost <= Value of your data

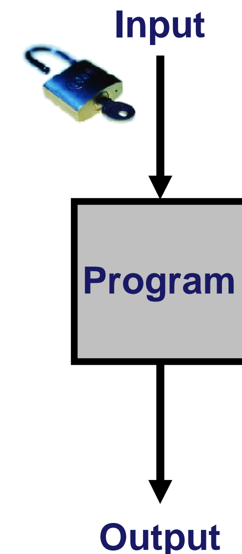
  - Therefore we should make the hacker’s goal more difficult
    - Better security systems
    - Less software errors

Distrusting input data

The code and the application

- **The code builds programs (applications)**
  - The computers are based on von Neumann model
  - They store the application code and its data in the same structure (memory)
  - Therefore the program may affect its own code
- **A computer program accepts data, processes it and returns result (output)**
  - It should conform to the program specification
- **The program input is the most crucial from the security point of view**

J. von Neumann (1903-1957)



Input data sanitization

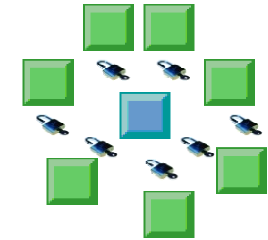
- **This is the most meaningful sentence I would like you to remember today:**

The most significant reason of software security bugs is a lack of (or insufficient) sanitization of the input data passed to the program

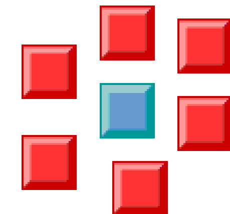
However, *input data* may mean:

- **Application input parameters…**
  - The most obvious way of passing data to the program
  - For standalone applications: the command line parameters
  - For Web applications: POST/GET data
- **… and also other (many) data sources!**
  - Environment variables
  - Configuration files
  - Output from the internal database (!)
  - Authentication data (e.g. X509 certificate DN)
  - The content of received network packets (including the data returned from external services)
  - The data entered interactively by the user
  - For Web application: cookies
  - …

I just get the filtered data…



- **Sometimes we write an internal function, accepting some data that already "should have been" filtered**
  - Our internal function may be copied to another module that does not assure sufficient sanitization
  - The function may work on another OS/hardware under different conditions
    - It was one of the causes of the Therac 25 incident
  - Someone might reuse only a part of the function, not aware about sanitizing issues



- **Indeed, it makes no sense that every single function has its own sanitization mechanism – but at least remember the problem**
  - NULL pointers may cause the most trouble

Bugs in standalone applications

An overview

- **C/C++ applications**
  - Buffer overflow + real example
  - Race condition (TOCTOU) + real example
  - Memory leaks + real example
  - Lack of verifying the return value + real example
- **All examples are taken from PSNC Security Team work on gLite software or EGEE monitoring services**
  - Well… almost all – but the others are worth mentioning!

Buffer overflow (1)

- **Introduction**
  - Beware of statically sized buffers!
  - 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 lead to memory protection fault
    - Specifically crafted data compose an exploit (a jump to the code specified by the attacker)
  - Especially root and suid root applications are dangerous
- **There are several kinds of buffer overflows**
  - On the stack (static buffers)
  - On the heap (dynamic buffers)

Buffer overflow (2)

- **Example 1**

```
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 = CA_MAXPATHLEN + 27 bytes
  - The logbuf buffer is able to contain CA_MAXPATHLEN + 8 bytes
  - It is possible to overwrite up to 19 bytes in memory (on the stack)

Buffer overflow (3)

- **Example 2**

```
line_buf = malloc (sizeof (char) * 1024)
...
for (i = 0; i < bufsize; i++, j++)
{
  if (buffer[i] == '\n')
  ...
  else
  {
    line_buf[j] = buffer[i];
  }
}
```

- **Explanation**
  - line_buf may store up to 1024 bytes (1023 + '\0')
  - bufsize denotes the length of the config file (max. 100000 bytes)
  - A line longer than 1024 bytes will cause heap overflow!

Buffer overflow (4)

- **Avoiding buffer overflows**
  - Be extremely careful when operating on local, statically sized buffers
  - Always calculate the maximum possible size of the buffer contents
    - Consider dynamic allocation of the buffer (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 those that should be handled by library functions

Buffer overflow (5)

- **The „non-developer" countermeasures**
  - 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
- **Or maybe use another programming language?**

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
    - Something might happen before it is returned to our code
  - It happens that the operation seeming to be atomic, is not!
  - Take a special care when
    - First verifying the files and then opening them
    - Creating temporary files
    - Multiple reading the same external data, e.g. environment variables

Race condition (2)

- **Example**

```
205: if (getenv(GLITE_METADATA_SD_ENV))
206:     ret = _glite_catalog_init_endpoint(ctx,
  metadata_namespaces,
207:                    getenv(GLITE_METADATA_SD_ENV));
```

- **Explanation**
  - If, between executing lines 205 and 207, an attacker modifies the contents of the GLITE_METADATA_SD_ENV variable (e.g. undefines it), _glite_catalog_init_endpoint() may receive malicious data (e.g. unexpected NULL)
  - The compiler should optimize the calls, but there is no guarantee

Race condition (2)

- **Example 2 (non-EGEE one):**

```
if (access(strFileName, R_OK) != 0)
{
  exit(1);
}
fd = open(strFileName, O_RDONLY);
// process the fd...
```

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

Race condition (2)

- **Avoiding race codintion:**

```
char* strEnv = strdup(getenv(GLITE_METADATA_SD_ENV));
if (strEnv)
  ret = _glite_catalog_init_endpoint(ctx,
       metadata_namespaces, strEnv);
```

```
/*first drop privileges!*/
FILE hFile = fopen(strFileName, “r”);
if (hFile)
  //process the file strFileName...
```

Memory / resource leaks

- **Introduction**
  - A memory leak occurs when a dynamically allocated memory block is not freed
  - A resource leak occurs when a sort of resource (e.g. file handle) is allocated but not freed
    - In general, memory leak is a resource leak as well, but it makes sense to distinguish this class of bugs
  - Leaks that occur inside loops are especially dangerous
    - Even tiny leaks, repeated numerously, may exhaust the system resources
- **Threats**
  - Significant loss on application and system efficiency
  - In extreme cases – DoS on the application and system

Memory / resource leaks (2)

- **Memory leak example:**

```
394: buf = malloc (sizeof (char) * 256);
399: if (! subject_dn)
400: {
401: scas_log (0, "%s: Error: No subject DN found,
  this
402: element is mandatory\n", logstr);
403: return 1;
404: }
```

- **Explanation**
  – If the subject_dn is NULL, the function returns without freeing buf (256 characters)
  – If an attacker is able to call this function with malicious data (no subject_dn), may consume all memory available to the process

Memory / resource leaks (3)

- **Resource leak example:**

```
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);
    }
```

Memory / resource leaks (4)

- **Resource leak example – explanation**
  - 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

Memory / resource leaks (5)

- **Avoiding leaks**
  - Free dynamically allocated resources on each return path from the function
  - Use finally{…} or alike mechanism to assure that all allocated resources are released
  - C/C++: do not mix malloc()/delete and new/free()
  - 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 that appropriately to ease the live of your successors
  - Test the debug version for memory/resoure leaks
    - Own scripts / macros
    - Dedicated tools: BoundsChecker, Purify, Insure++, Valgrind

Verifying return values

- **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 will be said in “Dangerous Functions” part
  – Custom functions implemented by the developers
- **Threats**
  – Unexpected application behavior
  – Application crash (NULL pointer dereference)

Verifying return values (2)

- **Example 1**

```
234: routem = (struct routem *)
235:                 malloc(maxfd * sizeof(struct routem));
236: for (i = 0;i < maxfd;++i) {
237:     (routem + i)->r_where = invalid;
238:     (routem + i)->r_nl = 1;
239: }
```

- **Explanation**
  - If the memory allocation of routem structure fails, the line 237 will cause an application crash

Verifying return values (3)

- **Example 2**

```
402: E->AVal[ATok] = realloc(
403:                         E->AVal[ATok],
404:                         len);
405: if (E->AVal[ATok] == NULL)
406: {
407: return(FAILURE);
408: }
```

- **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

Verifying return values (4)

- **Recommendations**
  - 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:

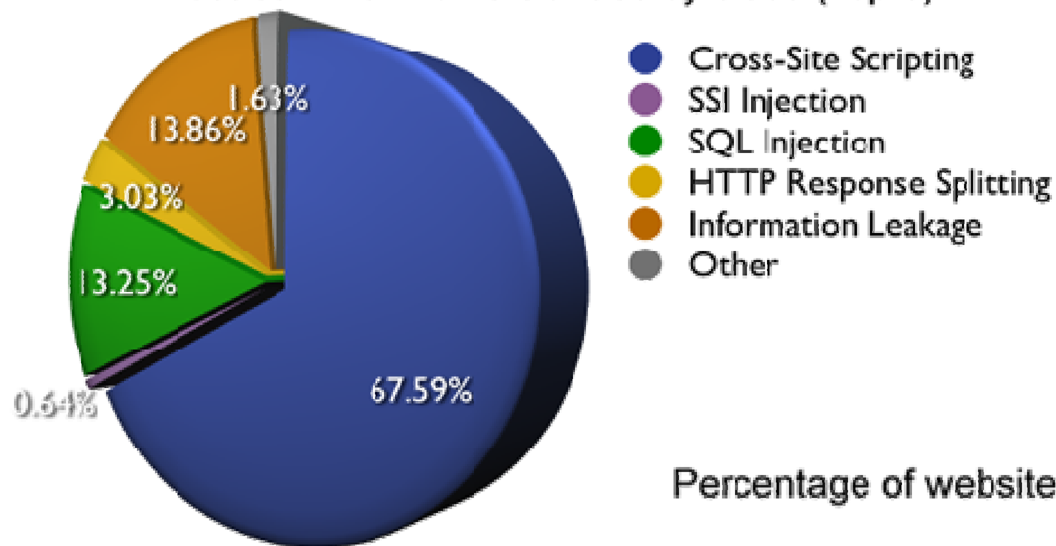
```
char *buffer, *temp;
int new_size;
buffer = malloc(1024);
if (buffer == NULL) exit(1);
new_size=2048;
temp = realloc(buffer, new_size);
if (temp == NULL) {
  free(buffer);
...
}
else buffer=temp;
```

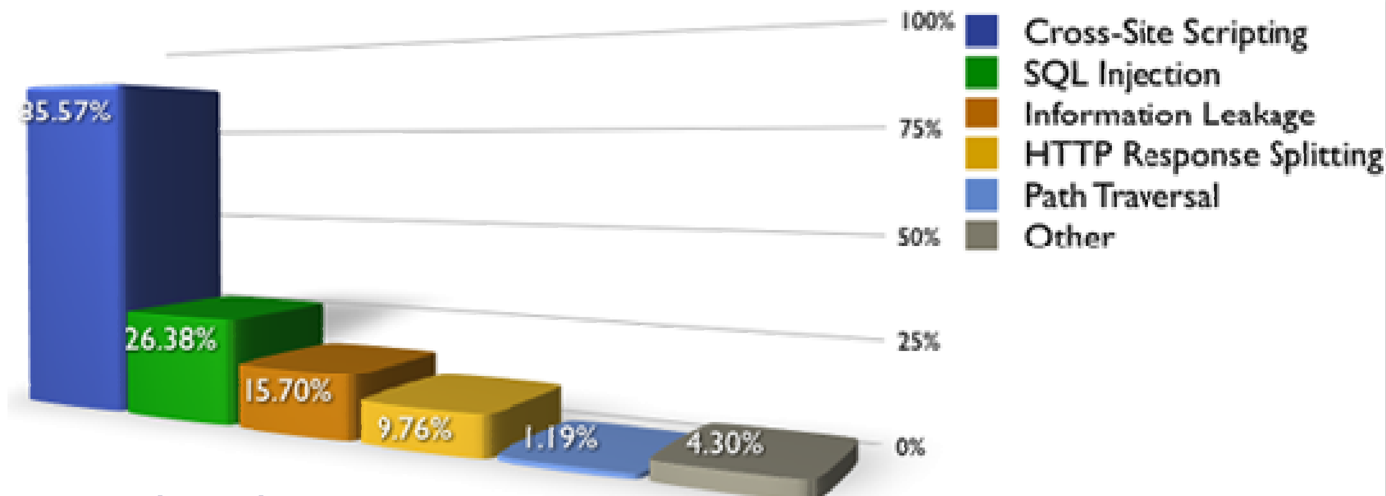
Bugs in Web applications

An overview

- **Web applications (PHP and Java-based)**
  - First some threatening statistics…
  - XSS + real examples
  - SQL Injection + real examples
  - Remote Code Execution + real examples
  - Directory Traversal + real examples
- **All examples are taken from PSNC Security Team work on gLite software or EGEE monitoring services**
  - Well… almost all – but the others are worth mentioning!

First some statistics…





Source: http://webappsec.org (2007)

XSS

- **Introduction**
  - Cross-Site Scripting is based on injection of the active (e.g. JavaScript) code in the content of a Web page
    - The victim displays a Web page, therefore executing the script in the context of his or her browser
  - The cause is a lack of (or insufficient) data filtering, especially those sent via GET and POST methods
  - Threats
    - Many people think there is no much harm, but…
    - Information disclosure (cookies)
    - Identity spoofing
    - Sophisticated attacks (e.g. scanning remote networks)
- **Remember that client-side data sanitization (although may be useful) is not enough!**
  - The above is valid for all kinds of Web application bugs

XSS (2)

- **Example 1**

```
$organization = $_GET['organization'];
$go = $_GET['go'];
...
else if($go==15 or $go==16)
{
    buttonback();
    echo"<h2>History for ".$organization."!!";
}
```

- **Explanation**
  - Passing malicious value in the URL might cause stealing cookies or invoking malicious activities
  - PoC:http://.../banner.php?go=15&organization=%3Cscript%3Ealert(document.cookie)%3C/script%3E

XSS (3)

- **Example 2**

```
String[] cols =
  m_schemaBrowser.getColumnStrings(tableName);
if (cols != null) {
...
} else {
  buffer.append("table name <b>" + tableName +
  "</b>");
  buffer.append("cannot be found in R-GMA.");
}
```

- **Explanation**
  - The value entered by the user in the URL (tableName) is directly attached to the final HTML code
  - PoC: http://<hostname>:8443/R-GMA/BrowserServlet/getQueryForm.do?tableName=<script>alert(1)</script>

XSS (4)

- **Example 3**

The blazej VO

REQUEST TO ADMINISTRATORS
REQUESTING VO MEMBERSHIP
LISTING REQUESTS
CONFIRMATION OF THE EMAIL ADDRESS

Virtual Organization Membership Service

Request to Administrators » requesting VO membership

VO User Registration Request

To access the VO resources, you must agree to the VO's Usage Rules. Please fill out all fields in the form below and click on the appropriate button at the bottom.

After you submit this request, you will receive an email with instructions on how to proceed. Your request will not be forwarded to the VO managers until you confirm that you have a valid email address by following those instructions.

IMPORTANT: By submitting this information you agree that it may be distributed to and stored by VO and site administrators. You also agree that action may be taken to confirm the information you provide is correct, that it may be used for the purpose of controlling access to VO resources and that it may be used to contact you in relation to this activity.

DN: /C=IT/CN=Tomasz Jakis/Email=tomasz.jakis@man.poznan.pl
CA: /C=IT/O=TEST CA
CA URI:
Family Name: Jakis
Given Name: Tomasz
Institute:
Phone Number:
Email:
comment:

I have read and agree to the VO's Usage Rules

I DO NOT agree to the VO's Usage Rules

XSS (5)

- **In order to join a VO a user had to fill the shown form**
  - The contents of the „Family name", „Given name" and „Institute" fields were not sanitized
  - After an e-mail confirmation, the new user's request appeared in „Request Handling" menu of the administrator view
  - Clicking on a pending request displayed the „Details of requests" page with the contents of the field above not sanitized
- **Account management operations were insufficiently protected**
  - Access to operations via "hidden" URLs
- **XSS attack on the form – removing an account**
  - **Family Name:**

    Miga<script>document.location="/voms/blazej/webui/request/admin/delete.do?reqid=25";</script>

XSS (6)

- **Avoiding XSS**
  - Never accept the data entered by the user without sanitization!
    - Especially distrust GET and HEAD variables
  - Be careful when building the HTML code using local database output, environment variables, X.509 certificates content etc.
  - Use regular expressions and/or functions like addslashes() or htmlspecialchars() for PHP
    - Disallow special characters like < > ; where just text is expected
    - Whitelist and regular expressions are a good approach
  - Use a simple scanner for detecting XSS (one will be described later)
  - Please note that PHP magic quotes feature is becoming deprecated and relying on it is discouraged!

- **Avoiding XSS - examples**

```
$organization =
  htmlspecialchars($_GET['organization']);

$organization = strip_tags($_GET['organization']);

$month = $_GET['month'];
if ($month == '1') $month='January';
elseif ($month == '2') $month='February';
...
elseif ($month == '12') $month='December';
else
  //error...
```

SQL Injection

- **Introduction**
  - SQL Injection is based on injection of arbitrary SQL code to an SQL query passed to the database by a Web application
    - An attacker uses malicious parameters and executes arbitrary query
  - The cause is a lack of (or insufficient) data filtering, especially those sent via GET and POST methods
  - An attacker may see the database output directly or indirectly (blind SQL Injection)
  - Threats
    - Information disclosure (database structure and contents)
    - Modifying the database contents
    - Deleting the database contents or the database itself
    - Remote code execution (via stored procedures, xp_cmdshell() etc.)

SQL Injection (2)

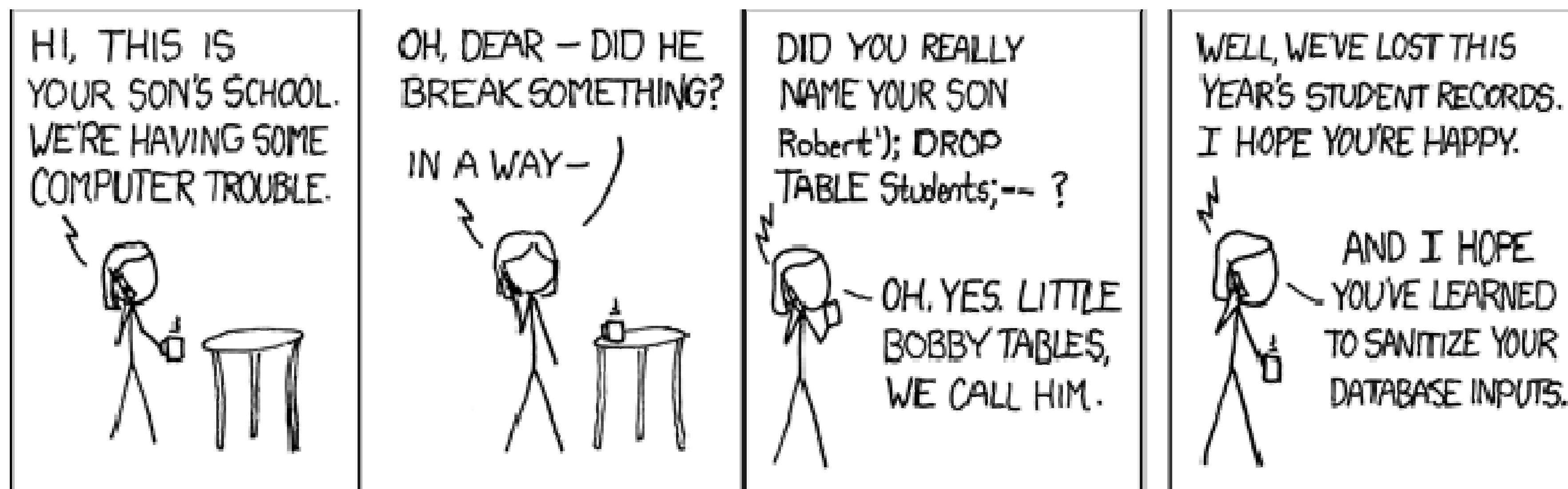
- **Example**

```
$sql="select \"Group\",\"JobStatus\",sum(\"NumJobs\")
from \"getSiteVoStNumJobs\"(".($t_end-
$value).",".$t_end.",".$_GET["filter1"].") GROUP BY
\"Group\",\"JobStatus\"order by
\"Group\",\"JobStatus\"";
//echo $sql."<br>";
$rs = pg_query ($conn, $sql);
```

- **Explanation**
  - The value entered by the user in the URL (filter1) is directly attached to the database query
  - PoC: http://.../gridice/test/jobs.php?filter1=11)%20group%20by%20"Group","JobStatus"%20union%20select%20NULL,NULL,NULL;--

SQL Injection (3)

- **Example 2 – non-EGEE one ;)**



Source: http://xkcd.com/327

SQL Injection (4)

- **Avoiding**
  - Basically, same as for XSS: insufficient (or a lack of) input data sanitization is the main problem
  - Using built-in mechanisms like parametrized queries instead of building query with string concatenation of user parameters
  - Using stored procedures
- **Non-developer countermeasures**
  - Applying least privileges principle for credentials definition
  - Appropriate structure of database
- **Please remember that the data you take from the local database also should be sanitized!**

Remote Code Execution

- **Introduction**
  - Sometimes Web application (e.g. PHP-based) use function calls that execute system commands (passthru, exec, system)
    - You should never allow your users to define their own arbitrary *commands*
  - The goal of the attacker is to craft the input data to be able to insert arbitrary system commands to be executed by the scripting engine
    - “A PHP console”
    - The commands are usually executed with the Web server credentials
  - Please do not confuse this chapter with executing arbitrary code as a result of exploitation of e.g. a buffer overflow

Remote Code Execution (2)

- **Example 1**

```
<?php
  ip=$_GET['ip']
  echo "Pinging host $ip"
  passthru (ping -c 4 $ip);
?>
```

- **Explanation**
  - The *ip* parameter is not sanitized
  - So why not apply a Unix shell meta-character ";"?
  - PoC:http://.../jobs/ping.php?ip=10.0.0.1;cat%20etc/passwd
  - Another (less critical) issue: why my Web server should offer remote scanning facilities to anyone?
  - Ping of Death: http://.../jobs/ping.php?ip=www.nasa.gov%20-c%209999%20-s%2065510

Remote Code Execution (3)

- **Example 2 (non-EGEE one, but worth mentioning)**
- **An HPC center website**
  - Users requesting for a computational account are required to send their CV, the list of publications could be also presented
  - The uploaded files were put in the *upload* subdirectory with a random name
  - The user was able to see his uploads on his or her account page (by referring to them directly by URL)
  - The extensions of the uploads were not verified
  - So why not upload a simple PHP script and call it a bit later?

```
<? $cmd = $_GET['c'];
   echo passthru("$cmd"); ?>
```

  - http://.../uploads/0f938...231ab94e8.php?c=cat%20/etc/passwd

Remote Code Execution (4)

- **Avoiding**
  - Analyze twice if this functionality is really necessary! Offer only as little functionality as required
  - Never allow the user to define his or her own *commands*, only specific parameters may be accepted
  - The parameters should not be just copied or concatenated
    - Try to use whitelisting for enumerated values and regular expressions for other
    - Remember not to relay on the client-side control!
  - Assure that all shell meta-characters are filtered out
    - Use PHP escapeshellcmd function
- **Non-developer countermeasures**
  - (PHP) use disable_functions in the initialization file to block functions like passthru, exec, system, shell_exec (will disable also the backtick operator (`)

Directory Traversal

- **Introduction**
  - Jumping out of the Web root directory, especially by providing ../ (directory up) characters in a parameter that builds a filesystem path
  - Important information may be disclosed, e.g. the contents of the system files
    - Usually the files are read with the Web server credentials
  - As usually, the causes and avoiding are associated with data filtering
    - Never do something like:

```
$content=$_GET["content"];
...
$file="content/".$content.".html";
readfile($file);
```

    - PoC: http://...vuln.html?content=../../../../../../etc/passwd%00

Directory Traversal (2)

- **Example**

```
$fileID = $_REQUEST['fileID'];
...
/* read all of the content of the file*/
$fullcontent = file ($tmpdir . $fileID);
```

  - $fullcontent is then processed and displayed on the page
  - PoC:
    - 1. Capture the request: GET /ldap/php/tree.php?actionID=expand&fileID=tmp/LEOO331mUA&row=2&... (tmp/LEOO331mUA name chosen by the application)
    - 2. Craft the request to GET /ldap/php/tree.php?actionID=expand&fileID=../../../../../../etc/passwd &row=2&...
    - 3. See the contents of the file: http://monitoring.egee.man.poznan.pl/ldap/php/tmp/LEOO331mUA

Other general issues

Handling sensitive data

- **Many applications (e.g. HYDRA) have to process sensitive data**
  - E.g. crypto keys, initialization vectors
- **These data must be stored somewhere in memory**
  - Another process may read this memory area
  - The sensitive data may be dumped to disk
  - If the developer is untidy, the data are stored in multiple (too many) memory areas
  - After the data are no longer necessary, must be deleted securely and as soon as possible
- **Memory disclosure attacks are not trivial, but still possible**
  - Read more e.g. at http://www.cs.utsa.edu/~shxu/dsn07.pdf

Handling sensitive data (2)

- **HYDRA issues reported**
  - In general, the sensitive data were treated only as solid as other (should be devoted more care)
  - Neither memset() nor custom procedures were applied to clear the buffers before deallocation
  - In several cases clearing or finalization functions applied to the sensitive data might be called a bit earlier
  - The keys/initialization vectors were duplicated, but not many times and it could be justified (bin and hex form of the key)
- **We feel that this issue is too rarely addressed**

Handling sensitive data (3)

- **Avoiding the problems**
  - Secure data deletion
    - memset() may be optimized by the compiler
    - implement ”secure memset”:

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

  - Consider using mlock() and/or mlockall() functions
    - Require root privileges
    - Lower the program efficiency

Handling sensitive data (4)

- **Avoiding the problems (cont.)**
  - Be extremely careful with realloc()
  - Minimize exposure
    - The number of duplications
    - The time between initialization and deletion
  - Use appropriate data types for storing sensitive data (should be store in mutable locations where they may be overwritten on demand)
    - e.g. in Java the char[] type should be rather used than String.
  - Non-developer countermeasures:
    - Disable core dumps e.g. via ulimit
    - grsecurity: set CONFIG_PAX_MEMORY_SANITIZE
    - Avoid hibernation

Dangerous functions

- **Every programming language hast its own list of „insecure” functions**
  – C should be especially mentioned – it gives more flexibility but with more potential errors
- **The dangerous functions should be used with care**
  – Sometimes it is not possible just to avoid them, they are necessary or helpful
- **During our tests in EGEE we addresed the following C functions the most often:**
  – strcpy/strncpy
  – malloc/calloc/realloc
  – strdup
  – sprintf/snprintf/vsnprintf

Dangerous functions (2)

- **strcpy/strncpy**
  - strcpy should never be used (and actually is not ☺)
  - strncpy is much better but does not repair everything
  - The destination buffer size should still be calculated properly
  - In some implementations, when the buffer will be exactly filled, will not be NULL-terminated
    - Always add 1 character for the terminating NULL
  - Be careful with multibyte characters!
- **strdup**
  - Remember that this function allocates memory that should be freed to avoid memory leaks
  - If the memory allocation fails, the function returns NULL, do not dereference this value without verifiying

Dangerous functions (3)

- **malloc/calloc**
  - Memory allocation may fail, you must properly handle such cases
  - Remember to free the allocated memory on all return paths
- **realloc**
  - The above rules apply as well
  - Additionally remember that the function does not clear the old buffer (significant for sensitive data handling)

```
char* newptr = malloc(NEW_SIZE);
memset(newptr, 0, NEW_SIZE);
memcpy(newptr, ptr, min(OLD_SIZE, NEW_SIZE));
secureMemset(ptr, 0, OLD_SIZE); /*not just memset()*/
ptr = newptr;
```

Dangerous functions (4)

- **sprintf/snprintf/vsnprintf**
  - Remember, what these functions return!
    - The return value is NOT the number of bytes actually written to the destination buffer
    - It is the number of characters that would have been written to the destination buffer if it was large enough to contain the whole formatted string

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

- Better:

```
int max_len = GLEXEC_MAX_LOG_LINE-
message_prefix_length;
int len = vsnprintf(buf+message_prefix_length,
max_len, fmt, ap);
message_body_length = (len > max_len ? max_len : len);
```

Coding conventions

- **Any convention that produces solid code is good ;)**
  - Although there are ones that make the code easier to read
- **Comments**
  - Function specifications
    - What does it do?
    - Input and output parameters
    - Caveats
  - Comments through the code are welcome as well
- **Remember that someone else may have to read and/or modify the code**
  - A security specialist will make more accurate opinion
  - It will be more difficult for another developer to make a mistake
    - Debian OpenSSL PRNG vulnerability (published in May 2008) was caused by commenting a line that „seemed” unnecessary by the Debian developers

Coding conventions (2)

- **Excellent function specification (glexec)**

```
/*!
 *  Check a string for the occurence of certain
 characters. This is specifically for
 *  the checking of environment variables that make it to
 a log file. The newline
 *  character '\n' is not allowed to appear in it as it
 allows reformatting the
 *  intended layout of the log file and may cause a
 potential exploitation.
 *
 *  \param variable  Variable to be checked
 *
 *  \return true, if the variable is found to be sane,
 false otherwise.
 *
 */
int glexec_sane_variable(const char *variable)
```

Coding conventions (3)

- **Excellent comments within the code (glexec)**

```
//  Check to see if the source proxy is not a link. Make
sure first to check
//  this, otherwise we'll check permissions etc. of the
link which will
//  fail for the wrong reasons.
if (S_ISLNK(stat_proxy->st_mode)) {
    glexec_log(GLEXEC_LOG_ERROR, "One of the input proxies
(%s) is a link and therefore rejected.\n", proxy_file);
    return GLEXEC_PROXY_IS_LINK;
}
```

- **On the other hand, avoid leaving old (commented) code**

```
/*
xacml_resource_attribute_t ra;
const char *resattr[2];
*/
```

We need more information…

More information

Secure Programming for Linu...

Table of Contents

1. Introduction
2. Background
   2.1. History of Unix, Linux, and Open Source / Free Software
      2.1.1. Unix
      2.1.2. Free Software Foundation
      2.1.3. Linux
      2.1.4. Open Source / Free Software
      2.1.5. Comparing Linux and Unix
   2.2. Security Principles
   2.3. Why do Programmers Write Insecure Code?
   2.4. Is Open Source Good for Security?
      2.4.1. View of Various Experts
      2.4.2. Why Closing the Source Doesn't Halt Attacks
      2.4.3. Why Keeping Vulnerabilities Secret Doesn't Make Them Go Away
      2.4.4. How OSS/FS Counters Trojan Horses
      2.4.5. Other Advantages
      2.4.6. Bottom Line
   2.5. Types of Secure Programs
   2.6. Paranoia is a Virtue
   2.7. Why Did I Write This Document?
   2.8. Sources of Design and Implementation Guidelines
   2.9. Other Sources of Security Information
   2.10. Document Conventions
3. Summary of Linux and Unix Security Features
   3.1. Processes
      3.1.1. Process Attributes
      3.1.2. POSIX Capabilities
      3.1.3. Process Creation and Manipulation
   3.2. Files
      3.2.1. Filesystem Object Attributes
      3.2.2. Creation Time Initial Values
      3.2.3. Changing Access Control Attributes
      3.2.4. Using Access Control Attributes
      3.2.5. Filesystem Hierarchy
   3.3. System V IPC
   3.4. Sockets and Network Connections
   3.5. Signals
   3.6. Quotas and Limits
   3.7. Dynamically Linked Libraries
   3.8. Audit
   3.9. PAM
   3.10. Specialized Security Extensions for Unix-like Systems
4. Security Requirements
   4.1. Common Criteria Introduction
   4.2. Security Environment and Objectives
   4.3. Security Functionality Requirements
   4.4. Security Assurance Measure Requirements
5. Validate All Input
   5.1. Command line
   5.2. Environment Variables
      5.2.1. Some Environment Variables are Dangerous
      5.2.2. Environment Variable Storage Format is Dangerous
      5.2.3. The Solution - Extract and Erase
      5.2.4. Don't Let Users Set Their Own Environment Variables
   5.3. File Descriptors
   5.4. File Names
   5.5. File Contents
   5.6. Web-Based Application Inputs (Especially CGI Scripts)
   5.7. Other Inputs
   5.8. Human Language (Locale) Selection
      5.8.1. How Locales are Selected
      5.8.2. Locale Support Mechanisms
      5.8.3. Legal Values
      5.8.4. Bottom Line
   5.9. Character Encoding
      5.9.1. Introduction to Character Encoding
      5.9.2. Introduction to UTF-8
      5.9.3. UTF-8 Security Issues

- **This is the beginning of the contents of David A. Wheeler's "Secure Programming for Linux and Unix Howto"**
  - An excellent book, especially for the developers
- **How many issues we were not able to mention?**
  - We need more links…

Coding security

- **http://www.dwheeler.com/secure-programs**
  - David A. Wheeler's book as PDF and HTML
- **http://www.securecoding.cert.org**
  - A website containing many interesting information on secure coding in general and C, C++ and Java related issues in detail
- **https://buildsecurityin.us-cert.gov/daisy/bsi-rules**
  - Per-function check: the Coding Rules section contains about 100 descriptions of C and C++ functions (including Windows APIs) with their caveats
- **https://edms.cern.ch/file/926685/1/EGEE_best_practices.pdf**
  - EGEE security best practices prepared by our team ;)

Web application security

- **http://www.owasp.org**
  - Open Web Application Security Project
  - An exhaustive source of information, e.g.:
    - OWASP Code Review Project
    - WebGoat Project
- **http://phpsec.org/projects/guide**
  - An online guidance prepared by PHP Security Consortium for securing PHP-based applications
- **http://ha.ckers.org/xss.html**
  - XSS Cheat Sheet – numerous ways to bypass sanitization (a „negative" vision, check OWASP for the positive one
- **http://code.google.com/p/browsersec**
  - Not quite for the developers, but a great review of browser security by Michał "lcamtuf" Zalewski

Questions or comments

Thank you for your attention!