Secure coding training

Data sanitization

Gerard Frankowski, PSNC

Poznań, 22-23 June, 2010

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

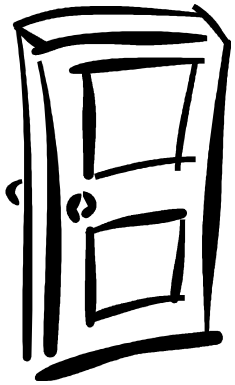
J. von Neumann (1903-1957)

Data filtering (sanitization)

- Security vulnerabilities may have different background
  - Inappropriate project (e.g. authentication data logged to a world-readable file, an error message reveals too much information)
  - Environment problems (e.g. vulnerable CMS)
  - Code-related problems (e.g. a buffer overflow, XSS)
- Today we speak about the last group
- **The main (if not the only) reason of such security vulnerabilities is insufficient, or a lack of, sanitization of the input data passed to the application**

What is input data?

- Application input parameters…
  - The most obvious way of passing data to the program
  - For standalone applications: the command line parameters
    - *C: void main(int argc, char ***argv[]**){*
    - *Java: public static void main(String[] **args**){*
  - For Web applications: POST/GET data
    - *http://www.application.com?param=value*
- … and also other (many) data sources!



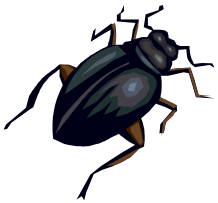
Other sources of input data

- 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 like DNS)
- The data entered interactively by the user
- For Web application: cookies, HTTP headers

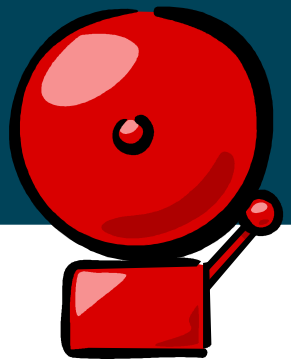
- Trust only the data defined inside your code!

Possible approaches

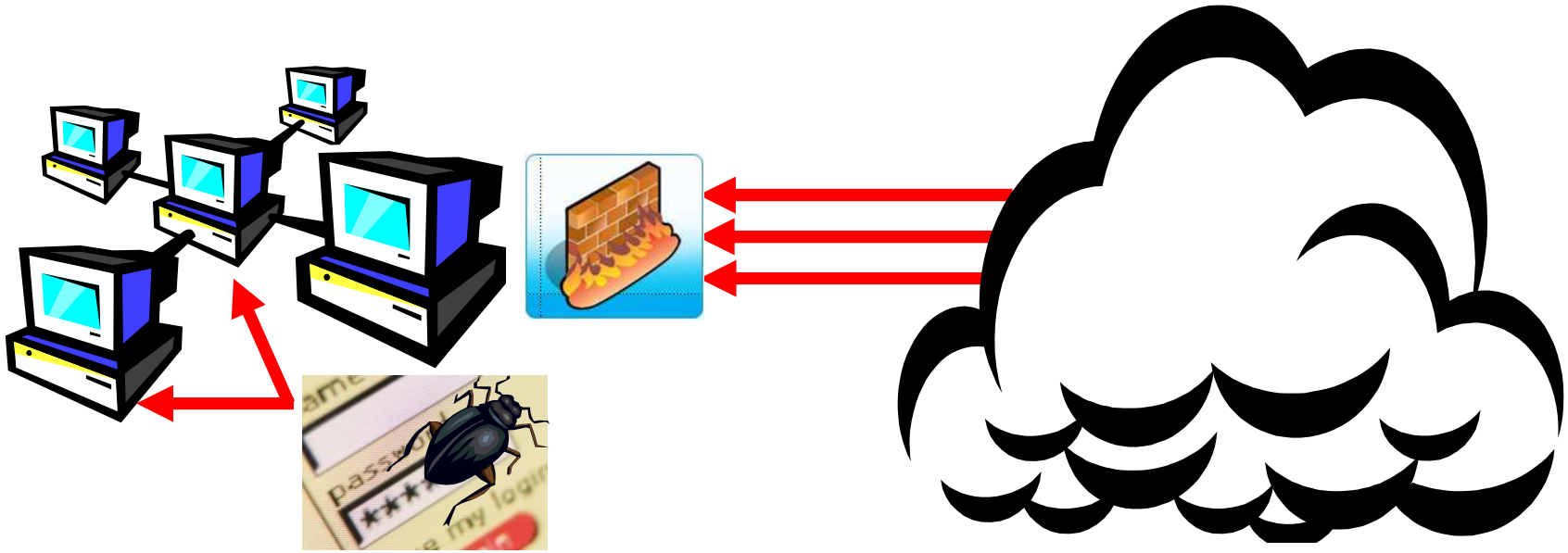
- Lack of data sanitization
- Filtering on the client side (esp. Web apps)
- White list approach
- Black list approach
- Regular expressions



Can I not filter at all?



- AAArgh…
- Why my application is so significant?



- So never, ever try that!

Filtering on the client side (Web application example)

- There are useful mechanisms to assure that the user does not enter out-of-range values
  - Tag properties

```
<input type="text" name="first_name" value=""
  size="20"/>
```

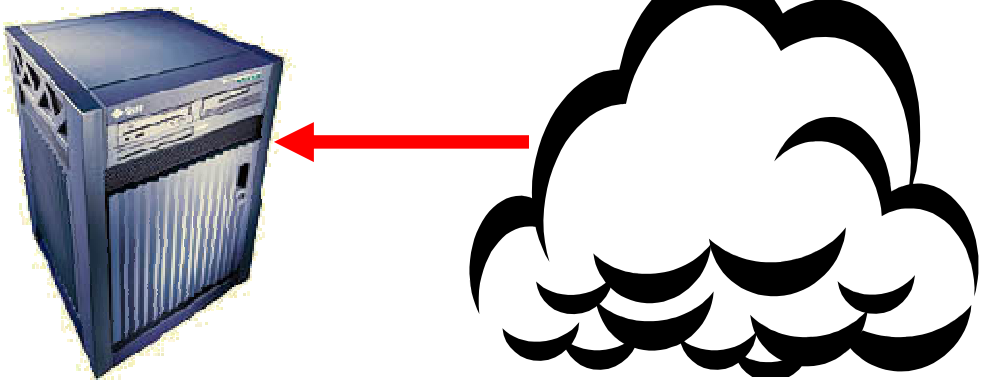
  - Validation functions

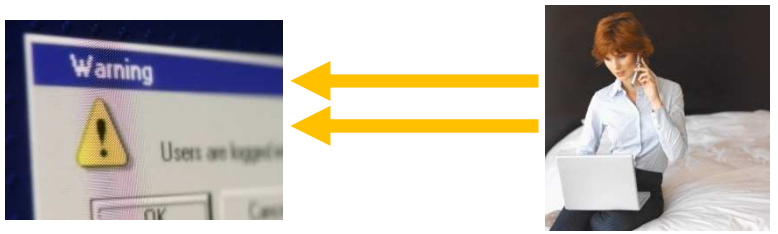
```
<form name="form1" action="./display.php"
  method="post" onsubmit="return validate();"
  />
```

- It is useful in terms of usability but **not** in terms of security!
  - Must not be the only protection layer!
    - *Actually, it is **not** a protection layer…*

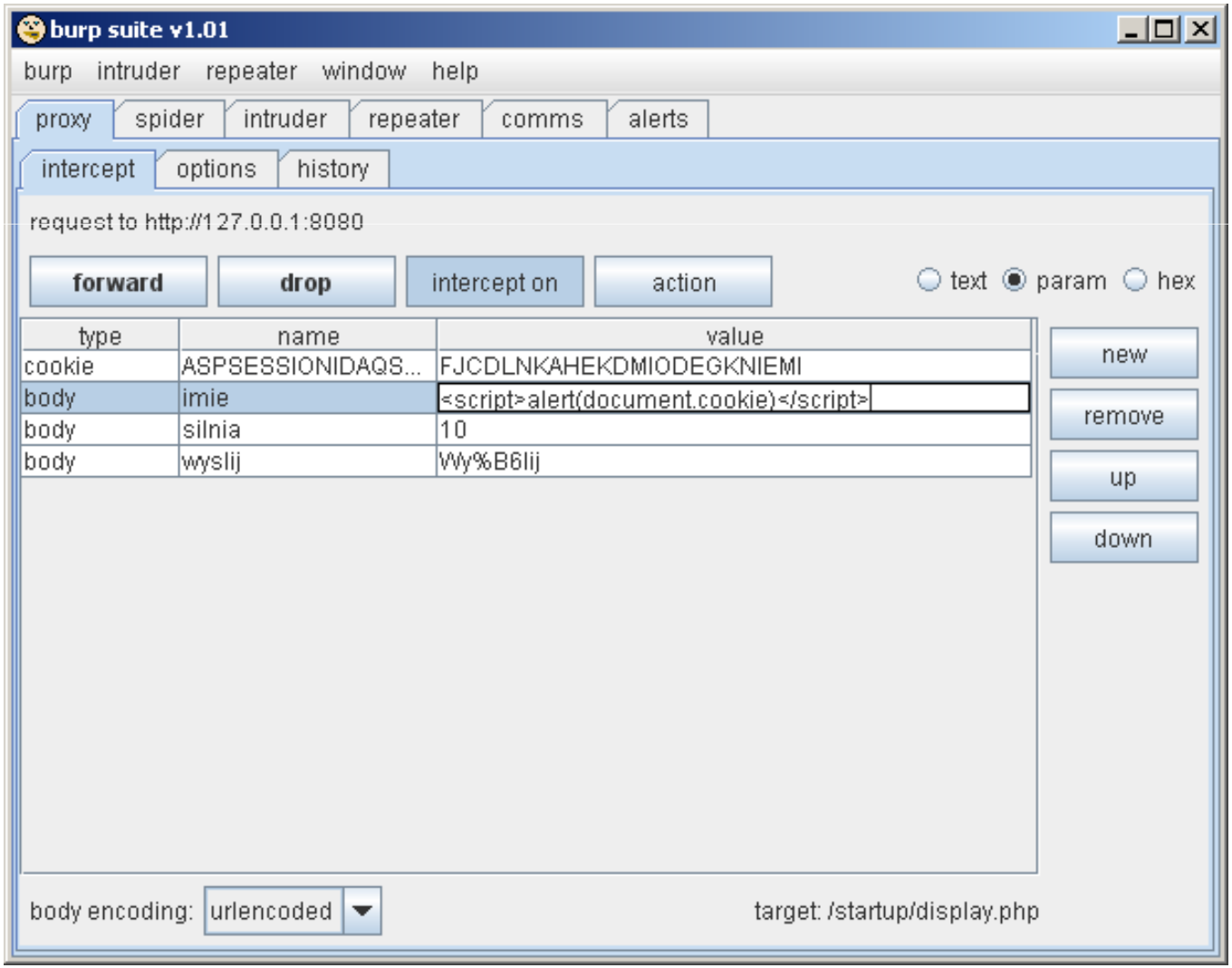
Filtering on the client side (2)

- Is your application the only way to contact the server?





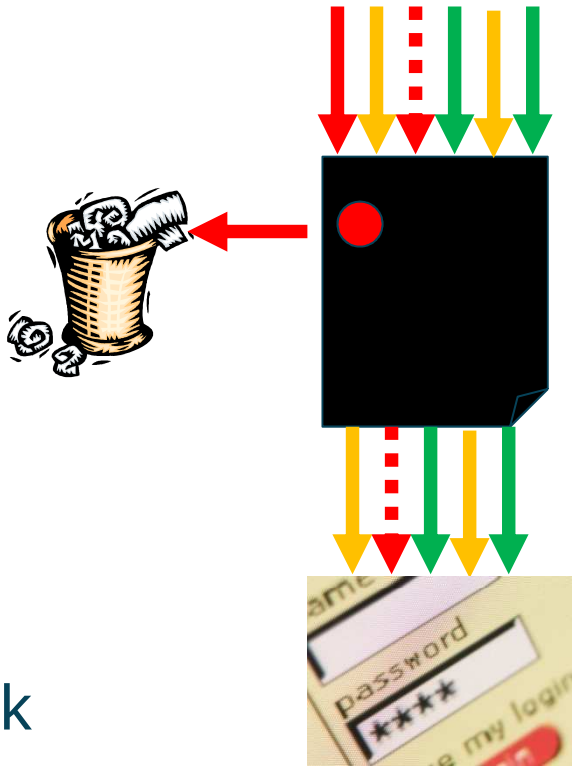
Burp proxy – a tool for easy sending crafted network packets



- http://portswigger.net

Black list approach

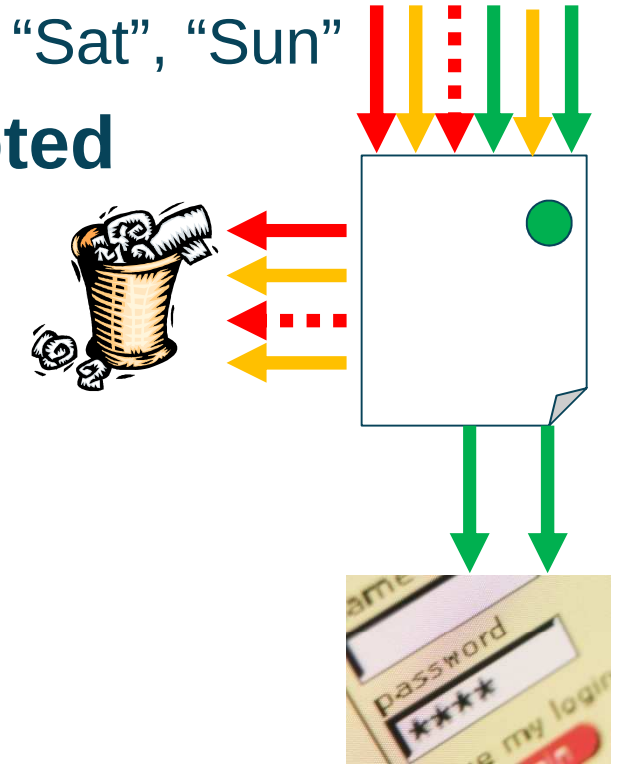
- You define what you **do not accept**
- e.g. “<script>”, “--”, “;”, “../”
- Everything that matches is **rejected**
- Everything else is **accepted**



- Advantages
  - Often much more simple to implement
- Disadvantages
  - Difficult or impossible to cover the attack vector
    - *New threats*
    - *e.g. %3CSCRIPT%3E…*

White list approach

- You define what you **accept**
  - e.g. “Mon”, “Tue”, “Wed”, “Thu”, “Fri”, “Sat”, “Sun”
- Everything what matches is **accepted**
- Everything else is **rejected**



- Advantages
  - Closer to the security principles
  - Much more precise filtering
- Disadvantages
  - The white list may be extremely difficult to define

Regular expressions

- A great help for definition of both black and white lists
  - Example: Polish postal code = ^[0-9]{2}-[0-9]{3}$

```
$strRegExp = “^[0-9]{2}-[0-9]{3}$”;
$strInput = $_GET[‘code’];
if (preg_match($strRegExp, $strInput))
  do_normal_processing();        //MATCH!
else
  do_error();                    //NO MATCH!
```

Whitelist

- A question: which type of the filtering list is implemented above?

Simplifying regular expressions

- The more tight the regular expression, the better
- However, in some cases the format definition might be too complicated
  - An example of “non-100% compliant” regular expression: email address (used for several years for registration for PSNC trainings)

```
^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$
```

- From security point of view, it is better to stop proper data than accept a malicious string
  - But ask your project leader!

Investigating the values

- Properly formatted data might be malicious as well
  - A (real) example: Web application accepts two values (startYear and endYear) and generates some statistics between the mentioned years

```
http://.../user/usnormal.php?query=usmaxmem&s
  tartYear=1000&endYear=2009
```

```
Fatal error: Allowed memory size of 16777216
  bytes exhausted (tried to allocate 71
  bytes) in /var/www/html/…/funciones.php on
  line 1047
```

- A more strict regular expression would help here too!
  - Sometimes it is easier to add some *ifs* than make sophisticated regular expressions

Summary

- Filter your input data at all!
  - Remember all sources of the input data
  - Implement the client side filtering, but for the user convenience, not for security
- Whenever possible, use white lists
  - Consider the structure of your data
  - Where it is too complicated, provide carefully designed black lists
- Use regular expressions for the format definitions
- Where applicable, additionally check the value