02230: Program Security

Robin Sharp

Informatics and Mathematical Modelling
Technical University of Denmark
Phone: (+45) 4525 3749
e-mail: robin@imm.dtu.dk

Basic Ideas

- A **program security flaw** is an *undesired program behaviour* caused by a *program vulnerability*.
- Work on program security considers two questions:
  - How do we keep programs *free from flaws?*
  - How do we *protect* computing resources *against programs with flaws?*
- Early idea was to *attack* the finished program to reveal **faults**, and then to *patch* the corresp. **errors**.
- Experience shows that this is not effective, and just tends to introduce new faults (and errors)!
- More modern approach is to use *careful specification* and *compare behaviour* with the expected.

IEEE Quality Terminology

IEEE Standard 729 defines quality-related terms:

- **Error:** A human mistake in performing some software-related activity, such as specification or coding.
- **Fault:** An incorrect step, command, process or data definition in a piece of software.
- **Failure:** A departure from the system's desired behaviour.

Note that:

- *An error may cause many faults.*
- *Not every fault leads to a failure.*

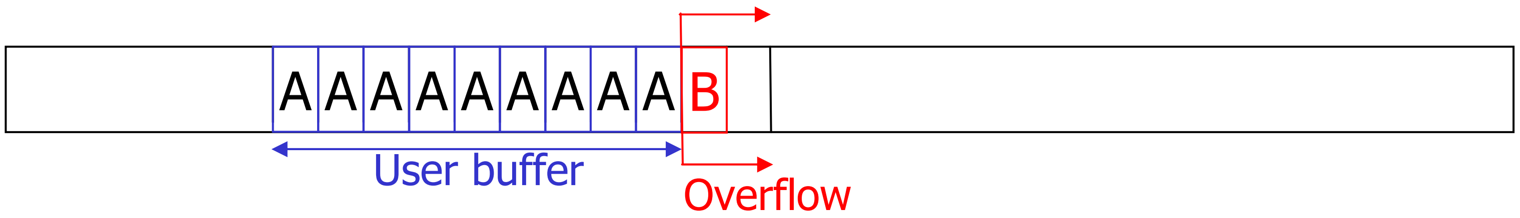
Program security flaws

Fall into two groups:

1. **Non-malicious flaws**. Introduced by the programmer overlooking something:
   - Buffer overflow
   - Incomplete mediation
   - Time-of-check to Time-of-use (TOCTTU) errors
2. **Malicious flaws.** Introduced deliberately (possibly by exploiting a non-malicious vulnerability):
   - Virus, worm, rabbit
   - Trojan horse, trapdoor
   - Logic bomb, time bomb

Buffer overflow

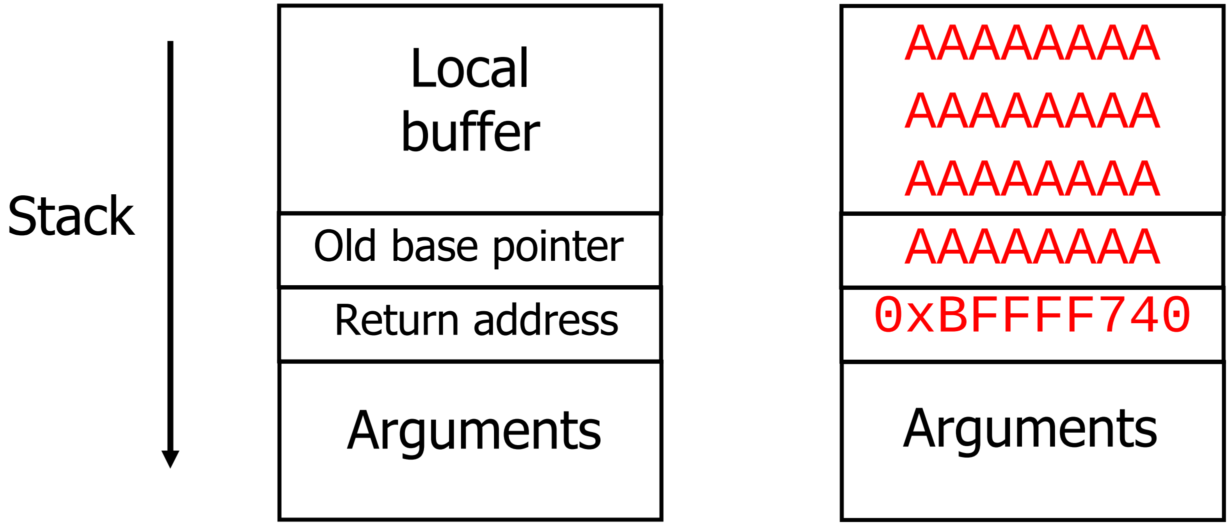
- A program that fails to check for buffer overflow may allow vital data or code to be overwritten:



- Buffer may overflow into (and change):
  - User's own data structures
  - User's program code
  - System data structures
  - System program code

Buffer overflow (2)

- Space for declared variables is in many languages allocated on the stack, together with return addresses.
- This means that overflow of a buffer can overwrite the return address:



Buffer overflow vulnerabilities

- String operations in C:

```
strcpy (dst, src);
strncpy(dst, src, sizeof dst);
```

`strcpy` unsafe, no checks that `dst` can contain `src`.
`strncpy` safe, but confusing (different from `strncat` etc.)

- Format string vulnerabilities in C:

```
printf("%s", buf0);
printf(buf1);
```

"%s" is format string, giving number and types of other args.
No checks that correct no. of args are in fact supplied.
So what happens if `buf1` contains the string "%s"?

Analysis tools

- **Static analysis** of program text:
  - ITS4 (Reliable Software Technologies/Cigital)
    http://www.cigital.com/its4
  - Flawfinder (Wheeler, 2001)
    http://www.dwheeler.com/flawfinder
  - LCLint/Splint (Evans et al. 2002)
    http://www.splint.org
  - Type qualifiers (Shankar et al., 2001)
  - Cyclone (Morissett et al., 2003)
- **Dynamic analysis** of execution:
  - Stackguard
  - Purify
  - CCured
  - Safe-C

Incomplete mediation

- Failure to perform “sanity checks” on data can lead to random or carefully planned flaws.
- Examples:
  - Impossible dates in correct format (say `yyyyMMMdd`):

    1800Feb30, 2048Min32

    What happens when these dates are looked up in tables in the program?
  - Alterable parameter fields in URL:

    `http://www.things.com/order/final&custID=101&part=555A&qy=20&price=10&ship=boat&total=`**`205`**

    Web site adds parameters incrementally as transaction proceeds. User can change them inconsistently.

Time-of-check to Time-of-use (TOCTTU)

- A **delay** between checking permission to perform certain operations and using this permission may enable the operations to be changed.
- Example:
  1. User attempts to write 100 bytes at end of file "abc". Description of operation is stored in a data structure.
  2. OS checks user's permissions on copy of data structure.
  3. While user's permissions are being checked, user changes data structure to describe operation to delete file "xyz".
- Can you find further examples?

Malicious code

- **Virus:** Attaches itself to program or data, passing malicious code on to non-malicious programs by modifying them.
- **Trojan horse:** Has non-obvious malicious effect in addition to its obvious primary effect.
- **Logic/time bomb:** Has malicious effect when triggered by certain condition.
- **Trapdoor/backdoor:** Gives intruder (possibly privileged) access to computer.
- **Worm:** Stand-alone program which spreads copies of itself via a network.
- **Rabbit:** Reproduces itself continually to exhaust resources.

Virus attachment

- Virus can attach itself to program or data by:
  - Appending itself, so virus code is activated when program is run. (Variation: Virus code before and after program.)
  - Integrating itself into program, so virus code is spread out over its target program.
  - Integrating itself into data, e.g. as an executable text macro.
- When activated, virus may:
  - Cause direct and immediate harm.
  - Run as memory-resident program, always available for use in discovering and infecting new targets.
  - Replace (or relocate) boot sector program(s), so malicious code runs when system starts up.

Virus detection

Anti-virus systems can be based on:

- Static analysis of code or data:
  - Look for **virus signatures**: characteristic patterns of instructions or data in files and/or memory.

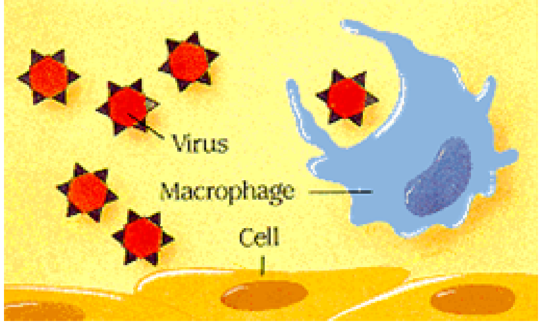
```
/default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
%u9090%u6858%ucbd3
%u7801%u9090%u6858%ucdb3%u7801%u9090%u6858
%ucdb3%u7801%u9090
%u9090%u8190%u00c3%u0003%uub00%u531b%u53ff
%u0078%u0000%u000a
HTTP/1.0
```

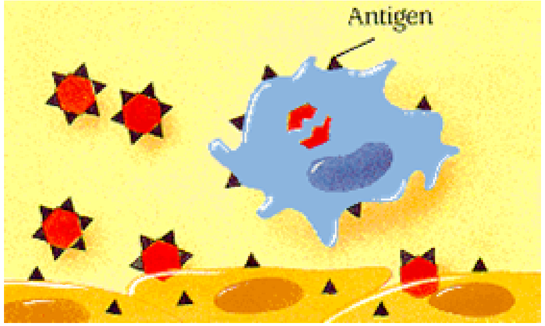
Code Red

- Dynamic analysis of behaviour:
  - Look for characteristic **behaviour patterns** (OS calls, etc.), for example by using Markov models, neural networks…

Immune systems

- In the human immune system, macrophages detect foreign proteins such as vira and “consume” them.
- This causes characteristic antigens to appear on the macrophage. These attract other white blood cells to attack and destroy the vira.
- Anti-virus systems in computers sometimes model these effects to attack “non-self”. (E.g. IBM anti-virus)





Covert channels

- A type of vulnerability which can be exploited to *access unauthorised information*.
- Analogous to **steganography**: transmission of information by hiding it in other information.
- Many techniques:
  - **Formatting** of data in output.
  - **Storage channels:** Information is passed via the state of objects in storage.
    a) Locking of a file (e.g. locked=1, unlocked=0)
    b) Existence of a file (e.g. yes=1, no=0)
  - **Timing channels:** Information is passed via the timing of events (e.g. short interval=0, long interval=1).
- The spy just needs to be able to "see" the channel.

Identifying covert channels (1)

- Covert channels depend on shared resources, so construct a matrix of resources vs. subjects:
- Look for rows/columns with the pattern :

	A	B
Resource 1	M	R
Resource 2	R	

- B cannot read from Resource 2, but A can pass info to B by reading Resource 2 and signal-ling by modifying Resource 1.
- So there is potentially info flow into the red box.

	Service process	Spy's process
Lock	Read, Modify	Read, Modify
Confidential	Read	Read

Identifying covert channels (2)

Denning's Information Flow method:

- Uses **static analysis** of program text based on syntax. For example: `B:=A` implies info flow A→B.
- Automatic analysis can reveal undesired info flows.
- Can be integrated into compiler or specification tool.

Statement	Flow
`B:=A`	A→B
`if C then B:=A`	A→B;C→B
`For k:=1 to N do stmts end`	k→stmts
`while k>0 do stmts end`	k→stmts
`case(exp) val1:stmts`	exp→stmts
`B:=fcn(args)`	fcn→B
`open file f`	—
`readf(f,X)`	f→X
`writef(f,X)`	X→f

Aims of program security

- Principal aim: Produce **trusted software** i.e. where code has been rigorously developed and analysed.
- Key characteristics:
  - **Functional correctness**: Program does what it is supposed to do.
  - **Enforcement of integrity**: Robust, even if exposed to incorrect commands or data.
  - **Limited privilege**: Access to secure data is kept to the minimum level necessary, and rights are not passed on to untrusted programs or users.
  - **Appropriate confidence level**: Program has been examined and rated to a degree of trust suitable for the data and environment in which it will be used.
- Obviously a product of good software engineering.