The Security Landscape & Controlling Computers

Today’s Plan

- **The Security Landscape (Elementary Information Security, Chapter 2)**
- Controlling a Computer
- Programs and Processes
- Buffer Overflow and the Morris Worms
- Access Control Strategies
- Keeping Processes Separate
- Lab 2: Learning Shell Scripting in Linux

The Security Landscape

The Security Landscape

- From security in a personal computer to problems in networks.

- Trade-off between security and benefits of ownership.

- Security choices fall into three categories:
  - **Rule-based decisions**
  - **Relativistic decisions**
  - **Rational decisions**

The Security Process

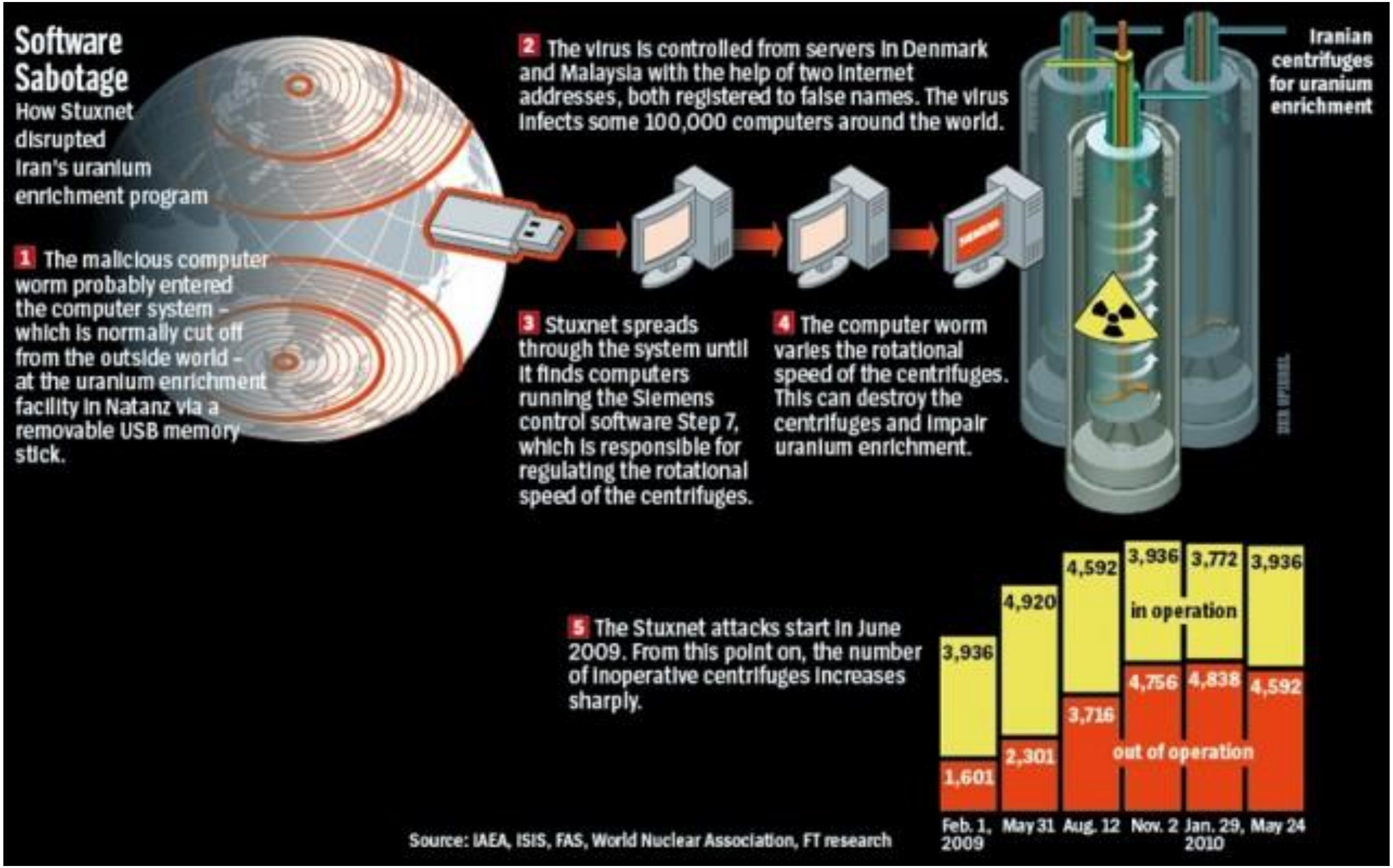
- Six steps:
  1. Identify your assets.
  2. Analyse the risk of attack.
  3. Establish your security policy.
  4. Implement your defences.
  5. Monitor your defences.
  6. Recover from attacks.

- Features shared:
  - Planning
  - Trade-off analysis

    } Early stages
  - Verification
  - Iteration

    } Later stages

- Moreover, it ensures that we have systematically reviewed the risks.

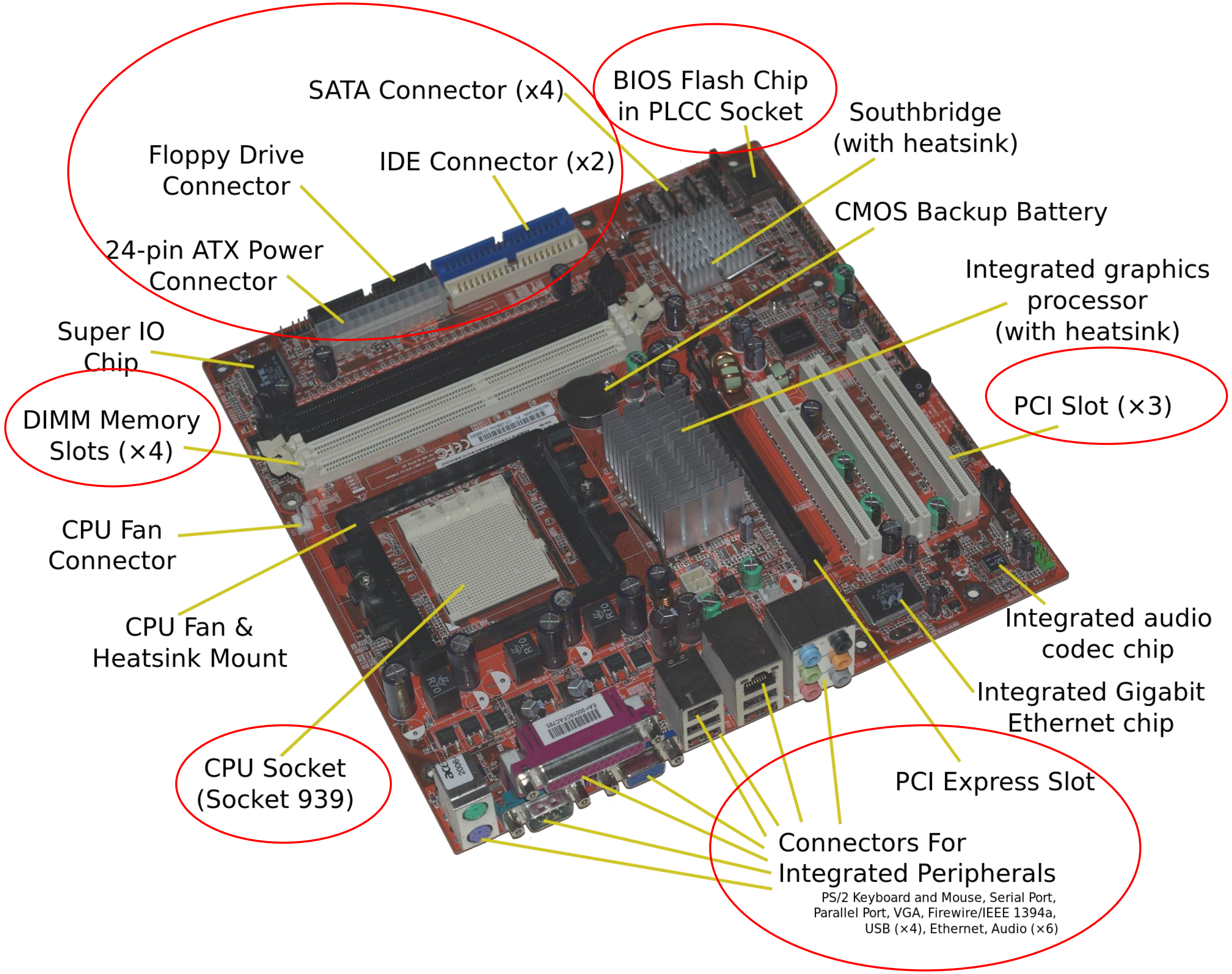


https://www.extremetech.com/computing/200898-windows-pcs-vulnerable-to-stuxnet-attack-five-years-after-patches

Controlling a Computer

Motherboard

1. CPU
2. RAM
3. BIOS
4. I/O Connectors



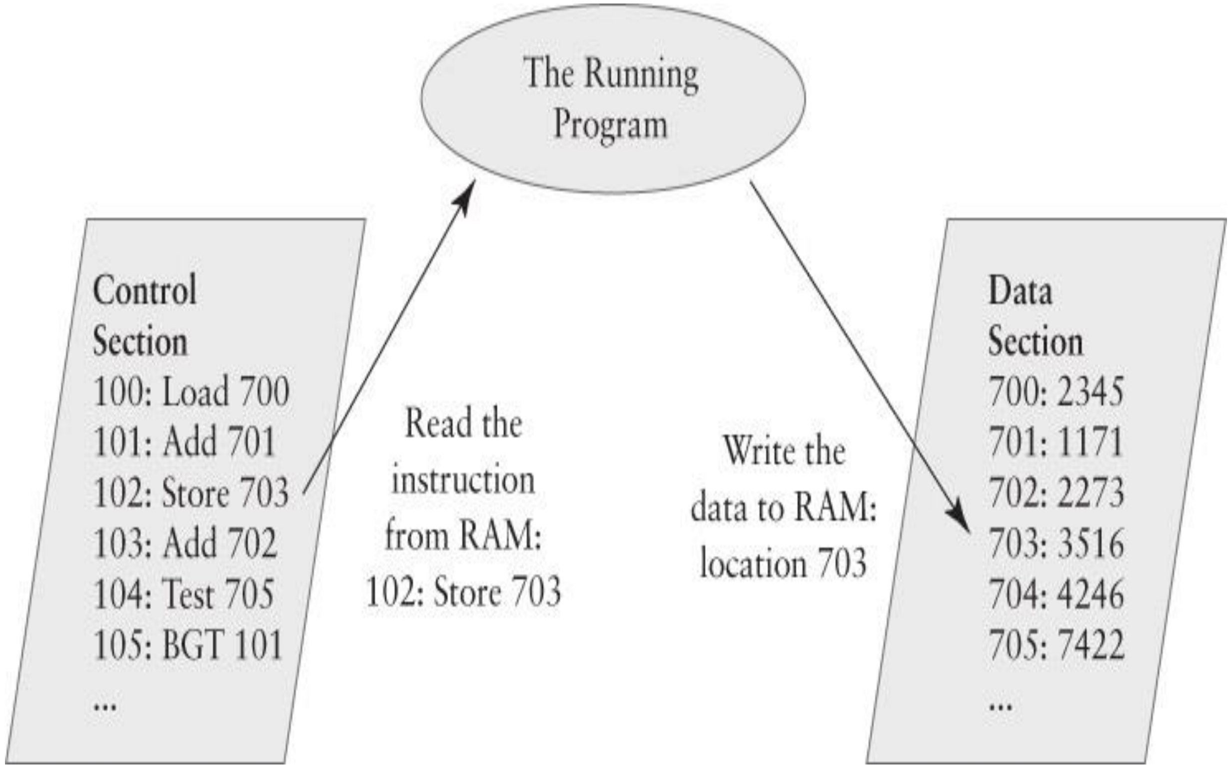
https://upload.wikimedia.org/wikipedia/commons/thumb/9/9d/Acer_E360_Socket_939_motherboard_by_Foxconn.svg/2000px-Acer_E360_Socket_939_motherboard_by_Foxconn.svg.png

Program Execution

- A program counter (PC) in the CPU keeps track of where a program's next instruction resides.
- To execute a program, the CPU:
  1. Retrieves the instruction from the RAM.
  2. Performs it.
  3. Updates PC to the point of the next instruction.

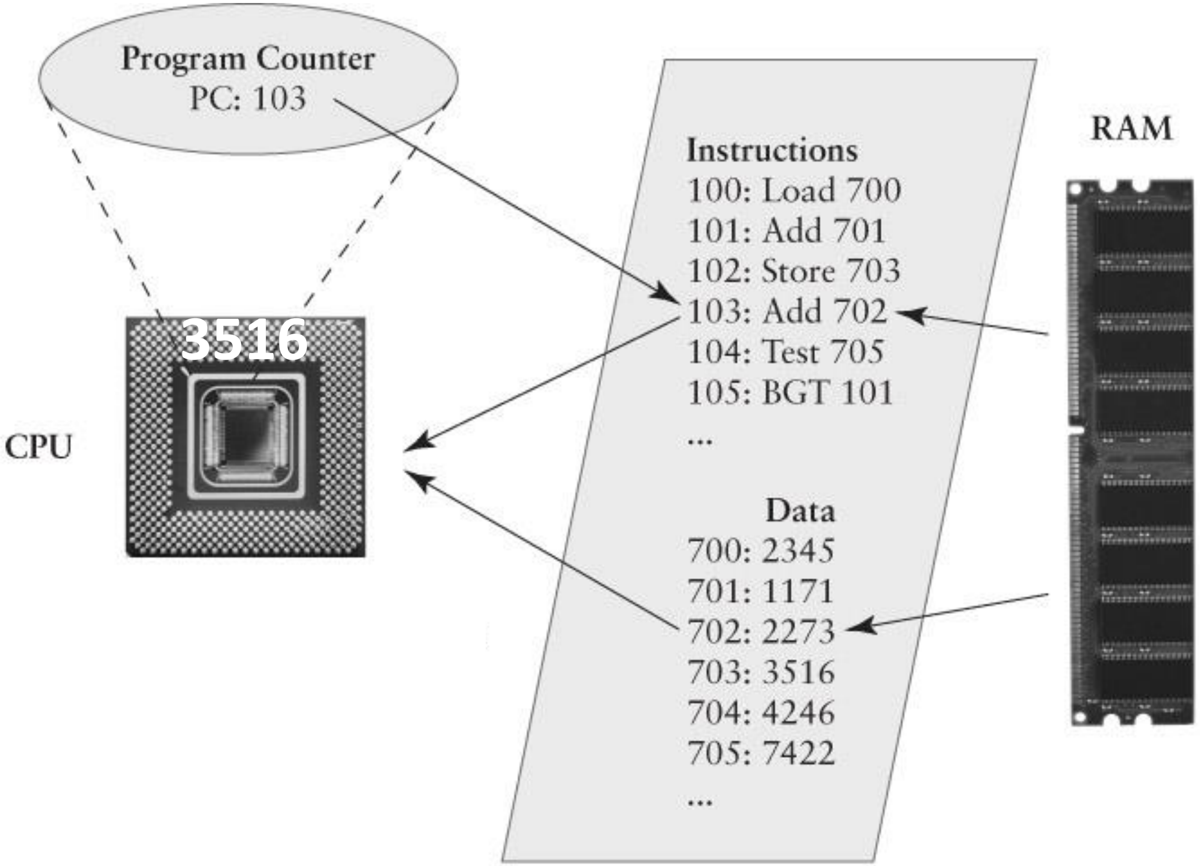
Program Execution

- RAM is divided into control and data sections.
- Access errors are due to programming errors or security issues.
- Each section is one byte (enough for a keyboard stroke but not for a command).
- Instruction must allocate enough space for address.
- Stacks and buffers also help.



Program Execution

After running instructions 100 (load 2345), 101 (2345+1171=3516) and 102 (store 3516 in data section 703), the state of a program is the following:



What happens next?

Programs and Processes

Operative Systems and Shells

- Typically a user wants to run more than one program at a time.

- The OS is in charge of interpreting the inputs (i.e. mouse clicks, keyboard strokes) and assigning them to different programs.

- Moreover, an OS such as Windows or Linux count with a **shell** (i.e. MS-DOS).
  - Meaning of MS-DOS?

- The shell has its own commands, which are executed directly by the OS.

Programs and Processes

- A program is a group of instructions.

- A process is a running program.
  - PC is, or can be, changing between processes.
  - Each process has some RAM with instructions and data.

- Windows example:
  - Run two command shells.
  - One program, two processes.
  - Looking at processes with the Task Manager (in MacOS/Linux, $*ps*).

Switching processes

- The “dispatcher” procedure in the OS switches between running processes.
- Stops (pauses) one process and starts another:
    1. Save the PC for the stopped process.
    2. Save other CPU data from the stopped process.
    3. Locate the “saved state” for the one to start.
    4. Load up the saved CPU data for the process.
    5. Load the PC with the starting process’ PC value.
- Parallelism illusion.

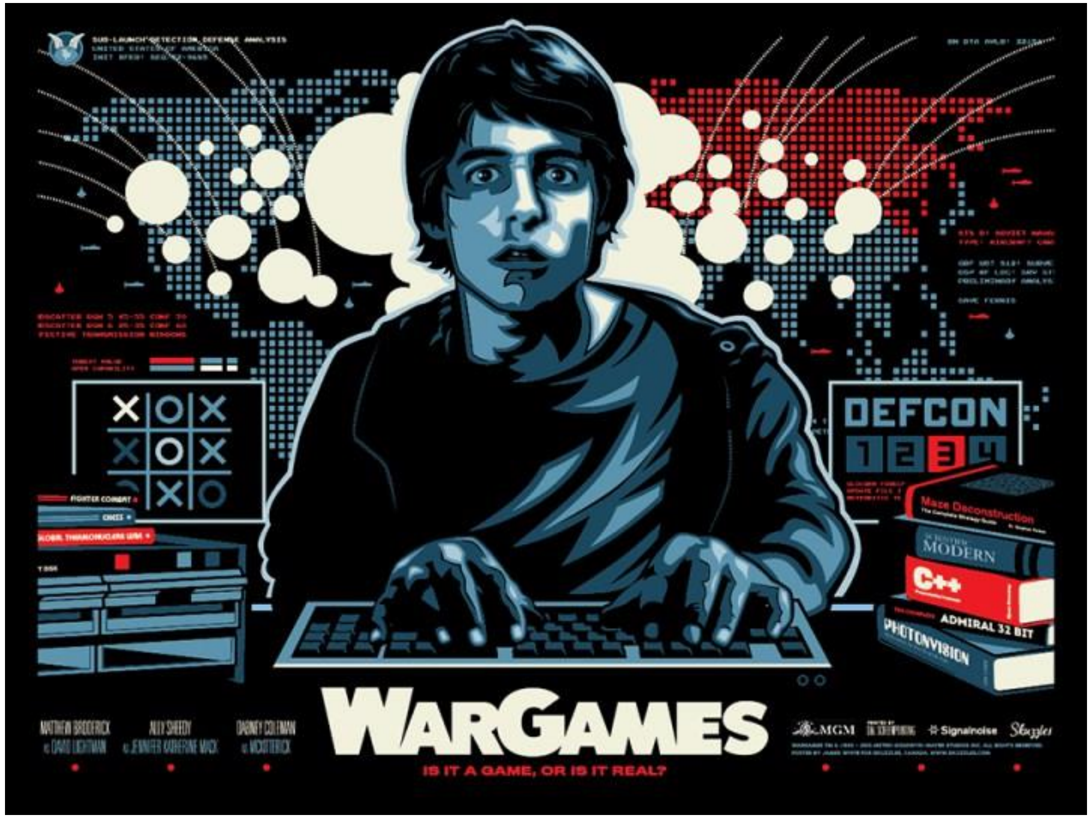
Buffer Overflows and the Morris Worm

Backdoors

- Cleverly designed messages that crash a software due to flaws in the network.
- Provide a connection by which an attacker can take control of the computer.
  - Simplest case: Control of the command shell.
- 1988: Robert T. Morris wrote a program that could replicate itself across computers connected to the internet (60’000 computers) by exploiting the *finger* service (Unix).

How could the worm fit on a floppy disk?

Mountain View, California



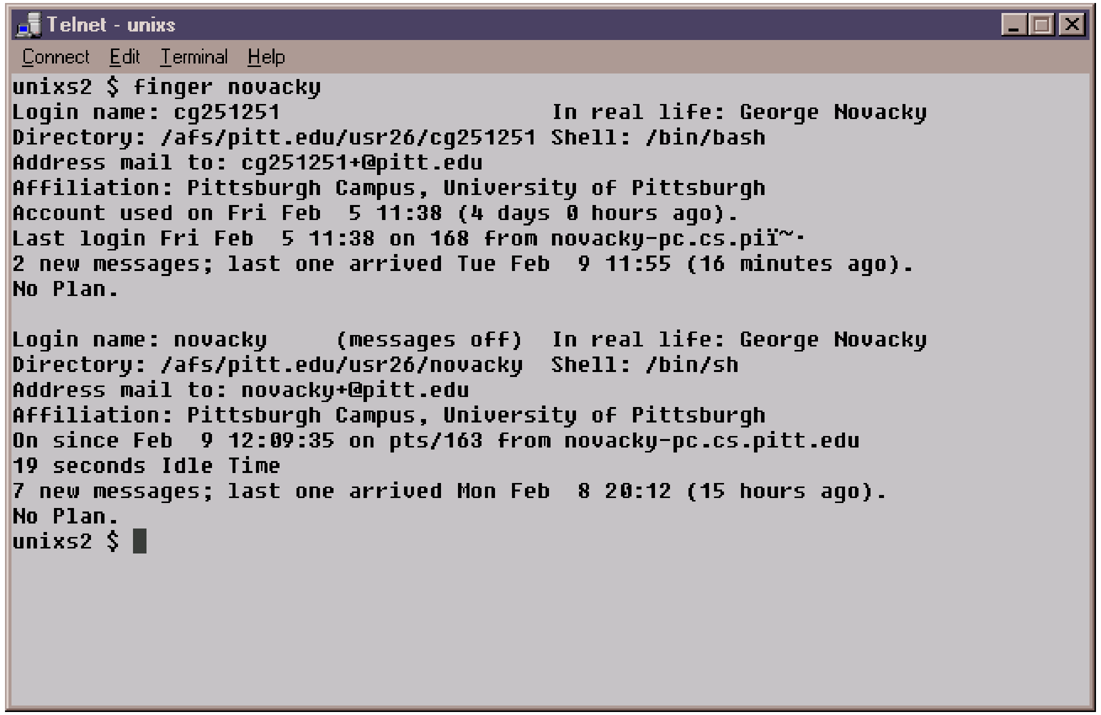
The Finger Protocol

- Purpose: Report status of individual computer users.
- Example: By typing `finger jsl@bu.edu`, the command starts a program that contacts the finger server *bu.edu* and asks about user *jsl*.
- Server responds if user is currently logged in, when last logged in, address, phone no., etc.

1020-1120: 100-byte **buffer** to store the user ID.

Finger didn't check the size in advance

"finger" Data Section
1000: return address to Network
1010: some finger variables
1020: buffer for user ID
1120: more finger variables
1180: return address to finger()
1200: some network "read" variables
1260: a network "read" buffer
1400: unused space
1450: unused space



Finger (Stack) Overflow

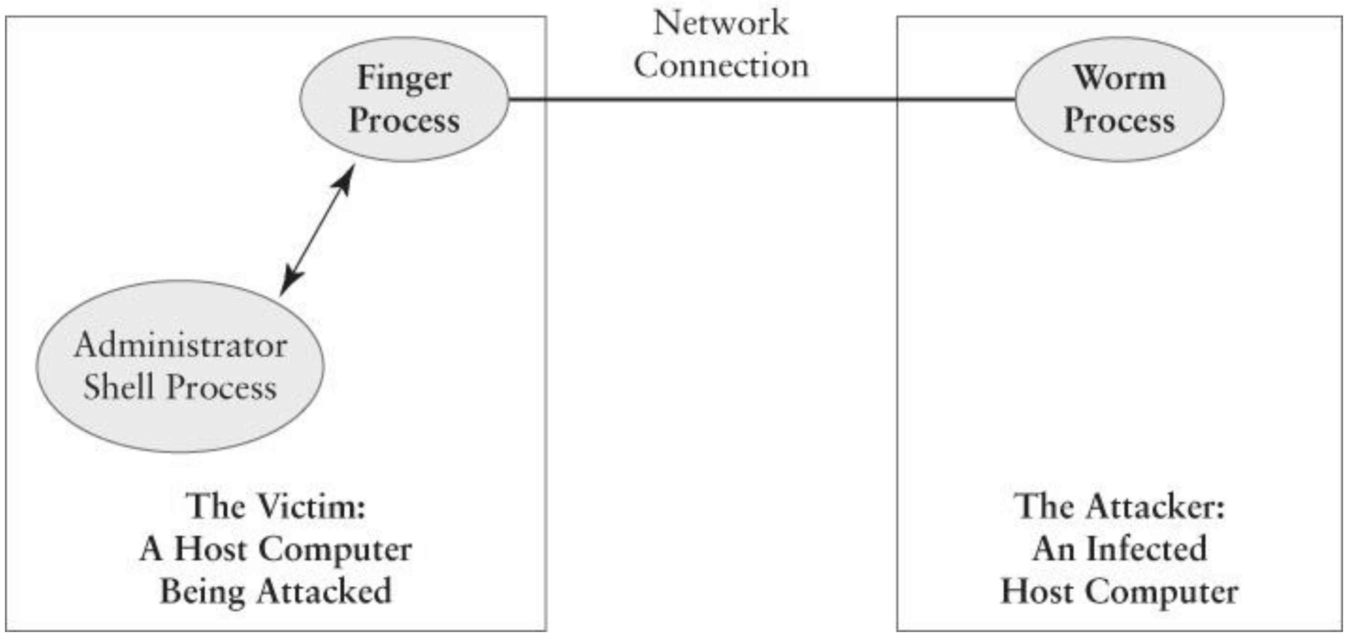
- If the sender typed to many letters, the program overflowed the buffer.

Normal

"finger" Data Section
1000: return address to Network
1010: some finger variables
1020: buffer for user ID
1120: more finger variables
1180: return address to finger()
1200: some network "read" variables
1260: a network "read" buffer
1400: unused space
1450: unused space

Malicious

"finger" Data Section
1000 return address to network
1010 some finger variables
1020 jsl@bu.eduXXXXXXXXXX
1120 XXXXXXXXXXXXXXXXX
1180 1260
1200 XXXXXXXXXXXXXXXXX
1260 [machine instructions that]
1400 ["take over" the finger process]
1450 XXXXXXXXXXXXXXXX



Overflow data content

- Understand how the finger service uses RAM to save PCs.
- The worm has two essential components:
    1. Sequence of computer instructions (shellcode).
    2. A storage location with the return address for the network *read* function.

“finger” Data Section
1000: return address to Network
1010: some finger variables
1020: buffer for user ID
1120: more finger variables
1180: return address to finger()
1200: some network “read” variables
1260: a network “read” buffer
1400: unused space
1450: unused space

Commands like “cmd” on windows to start shell unsuspectedly.

Once *read* finishes, it tries to return its caller by using the address stored at 1180, but the computer overflow rewrote that to 1260 (malicious code).

taken from attacking host computer

“finger” Data Section
1000 return address to network
1010 some finger variables
1020 jsl@bu.eduXXXXXXXXXX
1120 XXXXXXXXXXXXXXXXXX
1180 1260
1200 XXXXXXXXXXXXXXXXXX
1260 [machine instructions that]
1400 [“take over” the finger process]
1450 XXXXXXXXXXXXXXXXXX

Why this worked?

- CPU could not distinguish between instructions and data in the RAM.
  - Otherwise it wouldn’t execute the malicious shellcode instructions.
  - Microsoft has a **data execution prevention (DEP)** feature to avoid this.
    - It’s not default, applications have to specify it.
- The finger process ran with root privileges.
  - You could downgrade it, but it was “easier” this way (no trade-off analysis!).
- Most people used C, which doesn’t provide boundary checking.
  - Java automatically checks, but people still use C.
  - Modern C libraries allow buffer overflow check, but they are rarely implemented.

The Worm

- Released in October 1988.
- Promptly infected 10% of Internet computers:
  - The worm was designed to infect each computer once.
  - This restricting code did not work.
  - Each computer was infected hundreds of times!
  - Infected computers became unusable.
- Spread US-wide between 9pm and 11pm.

Fighting the Worm

- Telephone lines were not affected.
  - Analysts shared information by phone.
  - Many affected were meeting at Berkeley University.

- As computers were cleaned, they shared status and defensive data via email:
  - A ‘clean’ computer had to be hardened against the worm or it would be infected all over again.

Aftermath

- The worm incident helped create the Computer Emergency Response Team (CERT).
  - First US-wide, multi-organization computer security team.
  - Track and report problems.

- Today, reports are tracked by the Common Vulnerability Enumeration (CVE).

- Numerous public and private security organizations, like the “Internet Storm Center”.
  - Watch internet traffic and look for disruptions.

Access Control Strategies

For processes

Type 1: Islands

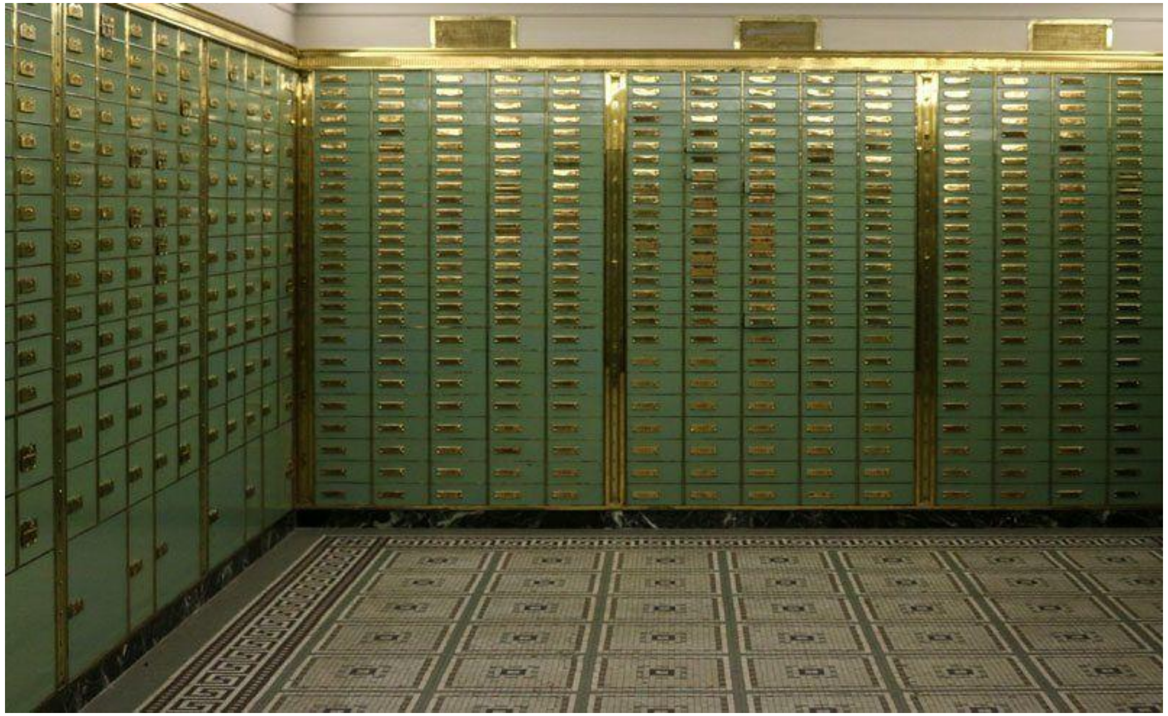
- Isolation and mediation.

- All processes begin in their own island.

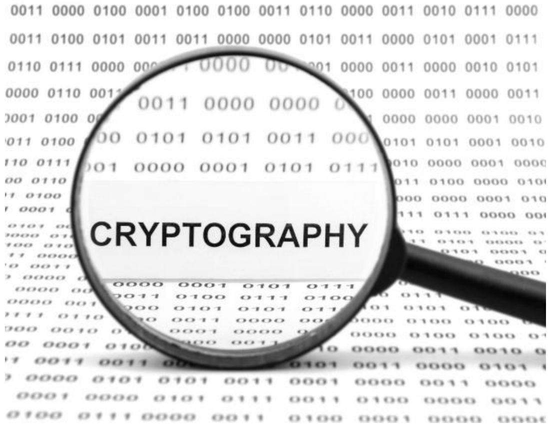
- Done especially on potentially hostile processes.

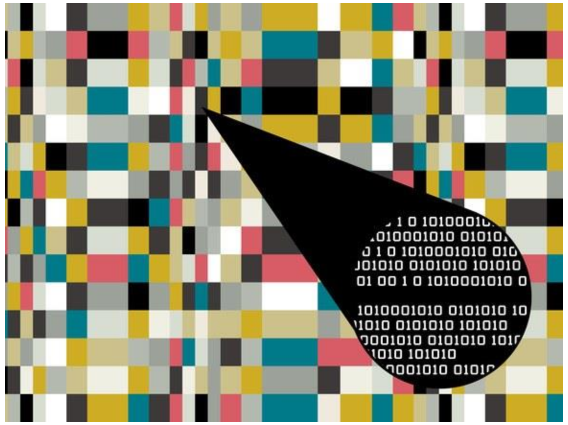
Type 2: Vaults

- Often OS provides access for processes (including “islanders”) to computer resources.
- Each request is checked by the OS.
- Access to RAM is of particular importance.



Type 3: Puzzles



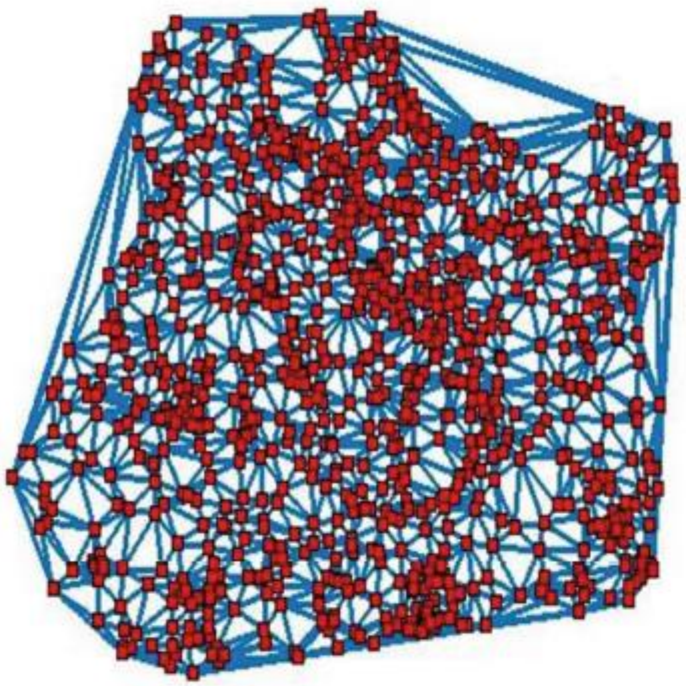


- Cryptography
- Steganography
- Security through obscurity (STO):
    - Hiding info (for good or bad purposes).
- *Cripto* becomes STO when easy to hack!
- *Steg* becomes STO when easy to look for!

Example of STO: Instead of addressing a security issue, threatening to sue the person that found it.

Type 4: Patterns

- Very common but unreliable.
- A computer antivirus recognises malware through comparison.
- Biometrics compare patterns.
  - Main problem: balance between false positive and false negative rate.
- Not effective against new threats.



Principle 1: Open Design

- Opposite of STO.

- Open the system for third-party analysis to help ensure its effectiveness:
  - “More eyes make bugs shallow” – Eric Raymond.

- Kerckhoff’s Principle and crypto-design:
  - The crypto system can be well-known, but NOT the key used!
  - “The enemy knows the system” – Claude Shannon.

Principle 2: Chain of Control

We must never run programs that violate or bypass our security policy. To avoid this, we:

1. Start the computer using a BIOS that maintains our security policy.
2. If the software we start (i.e. the OS) can start other software, then the other software either
   - Complies with the security policy, OR
   - Is constrained from violating the policy via access restrictions or other mechanisms.

Subverting the Chain of Control

- At the BIOS, attacker may:
  - “Boot” a different OS from a CD-ROM or a USB drive.
    - The new OS doesn’t enforce access restrictions.
- Inside the OS, attacker may:
  - Install a privileged (administrative) program that can bypass access restrictions.
  - Trick an authorized user into leaking sensitive files.
- Solution: Password protect BIOS user interface.
  - Worth it?

Keeping Processes Separate

Keeping Processes Separate

- Operator errors and software bugs also pose threats to the computer.

- Relies on hardware and software.

- Hardware by providing special features to the CPU:
    1. Two modes to distinguish between user and special (kernel) applications.
    2. RAM restriction.

- Software by including features in the OS:
    1. User identities
    2. Program Dispatcher
    3. Memory Manager
    4. RAM Protection
    5. Window-oriented interface

In a 90's OS, a damaged Word file could also damage an Excel spreadsheet. This was largely due to programs getting larger and occupying RAM registers which originally didn't belong to them.

Hardware Separation using Program Modes

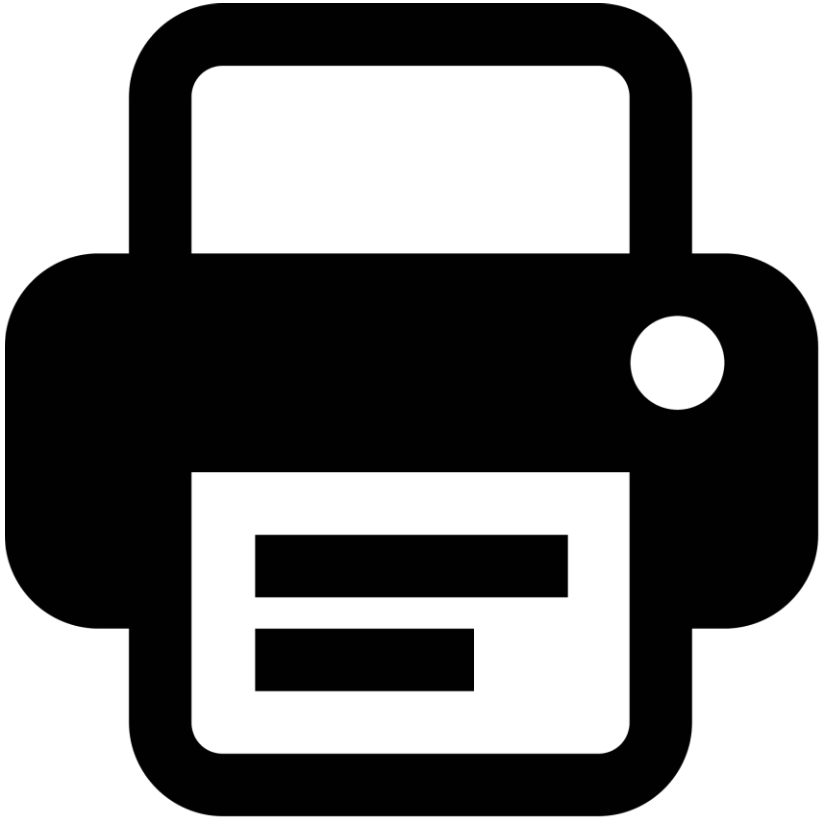
- Kernel or Supervisor Mode
  - For highly privileged operating system programs with full CPU access.
  - Allows full access to RAM.
  - Dangerous! Used as rarely as possible.
- User Mode
  - For most programs and all applications.
  - CPU blocks any attempt to use kernel mode instructions.

Hardware Separation using RAM Restriction

- Each process operates in its own RAM island implemented through a *page tabler* that allocates RAM (a table per process).
- Memory error when CPU finds a RAM register that doesn’t match the one assigned to the process.
- Page tabler also specifies if the process is allowed to read/write or only read in a specific RAM register (CPU can read/write any register in kernel mode).

Software Separation through User Identities

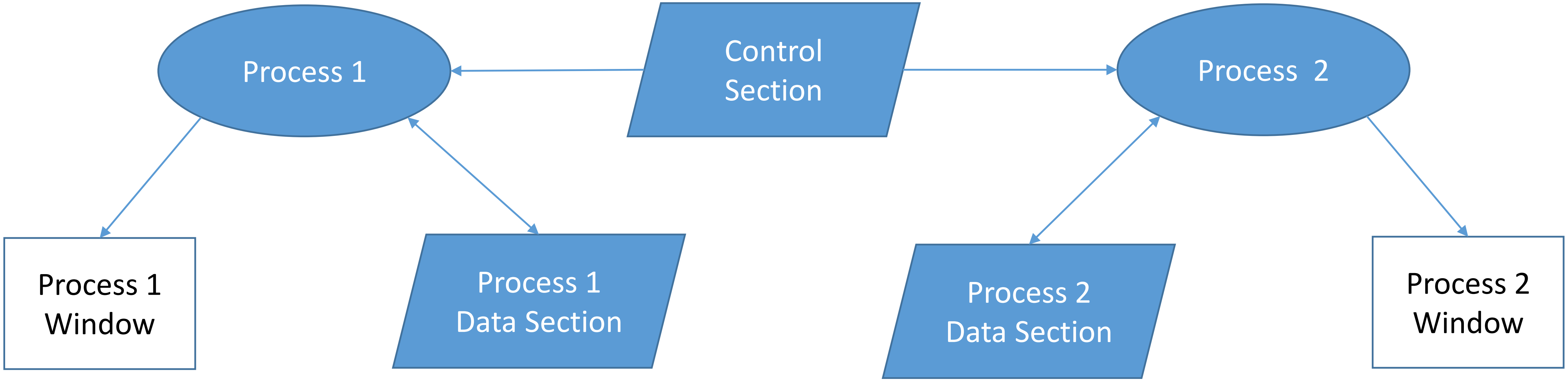
- OS provides the vault for each process.
- Users are assigned a user ID.
- Each user ID has specific resources (i.e. printer).
- OS checks the permissions before granting access.



Any Exceptions?

- Memory saving: Two processes running the same program.
- Networking software: Several processes pass network data using the same RAM buffer.
- Sharing data between processes (data section only).
- If one process changes the control section while the other is running, a problem can occur.
- Solution: OS restricts access to the control section. This is known as a **read only** access restriction.
- Access rights are best represented through an **Access (Lampson's) Matrix**.

Example 1



Control/Data Section	Process 1	Process 2	OS
Control Section	R-	R-	RW
Process 1 Data Section	RW	--	RW
Process 2 Data Section	--	RW	RW

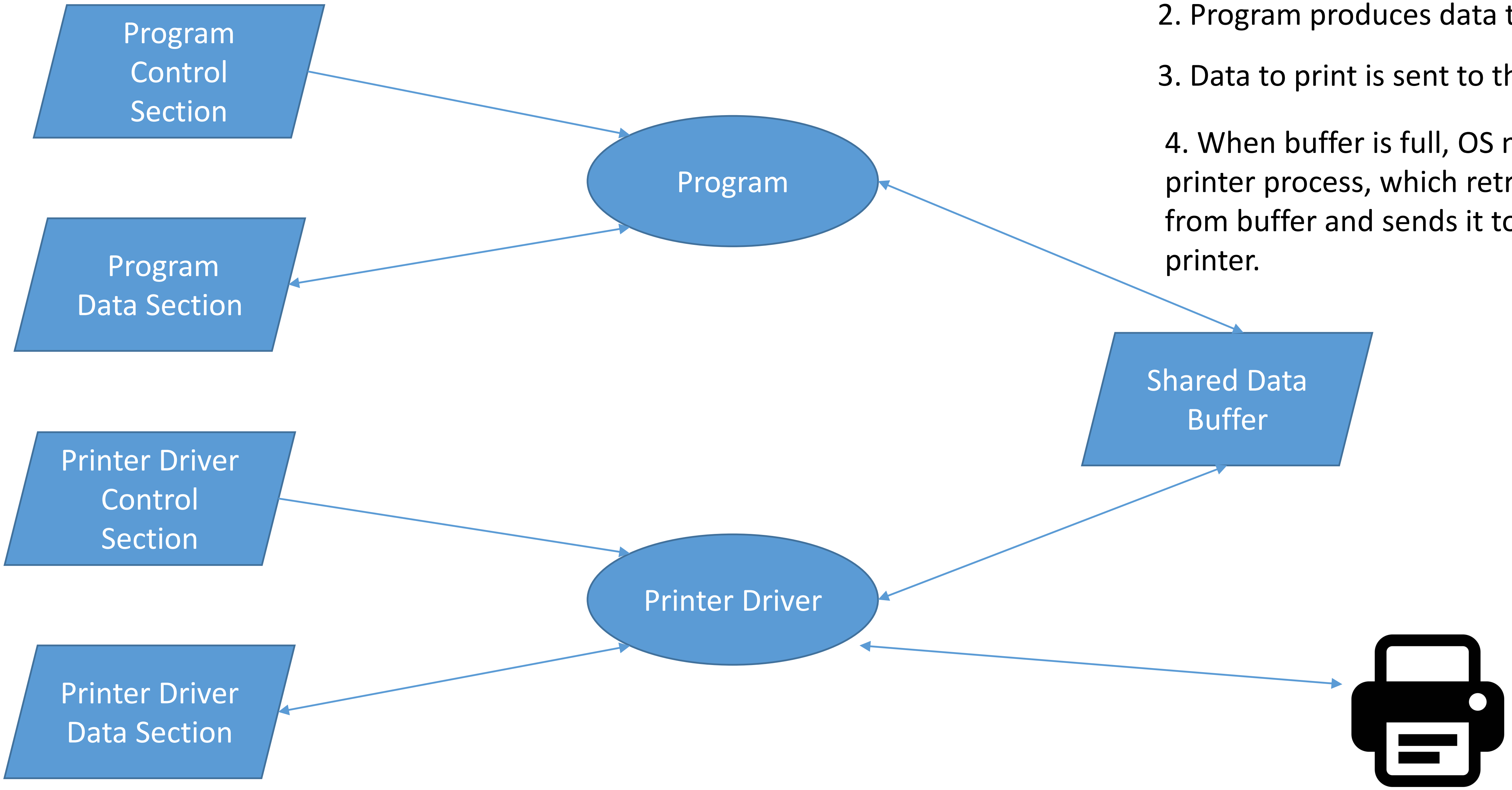
Sharing Data

- How to allow two processes to share data stored in RAM?
  - Normally processes are isolated from each other.
  - This prevents one process from damaging the other one.

- OS provides a separate data section:
  - Processes still have exclusive access to own data.
  - All shared data resides in this separate section.
    - Both processes have RW access to the shared section.

Example 2: Sending Data to Pinter

1. OS acquires a data section to share.
2. Program produces data to print.
3. Data to print is sent to the data buffer.
4. When buffer is full, OS notifies printer process, which retrieves data from buffer and sends it to the printer.



Access Matrix Example 2

Control/Data Section	Program	Printer Driver	OS
Program Control Section	R-	--	RW
Program Data Section	RW	--	RW
Driver Control Section	--	R-	RW
Driver Data Section	--	RW	RW
Shared Data Buffer	RW	RW	RW

- Driver is only granted the access it really needs.
- Blue screen: Windows finds a fatal error and displays text mode and prints CPU and RAM contents.
- Many of these problems result due to faulty device drivers (people who write drivers rarely know OS rules).

“Lab” 2: Learning Shell Scripting in Linux

- Go to linuxcommand.org.
- Go through section *Learning the Shell.*
  - *To test the commands, you can use your VM!*
- Further information and practice can be found on “The Linux Command Line” book (PDF is on Moodle).

Lab 3: Writing Shell Scripts

ADDITIONAL PRACTICE

- *Writing Shell Scripts in* linuxcommand.org.
- “The Linux Command Line” book (PDF is on Moodle).