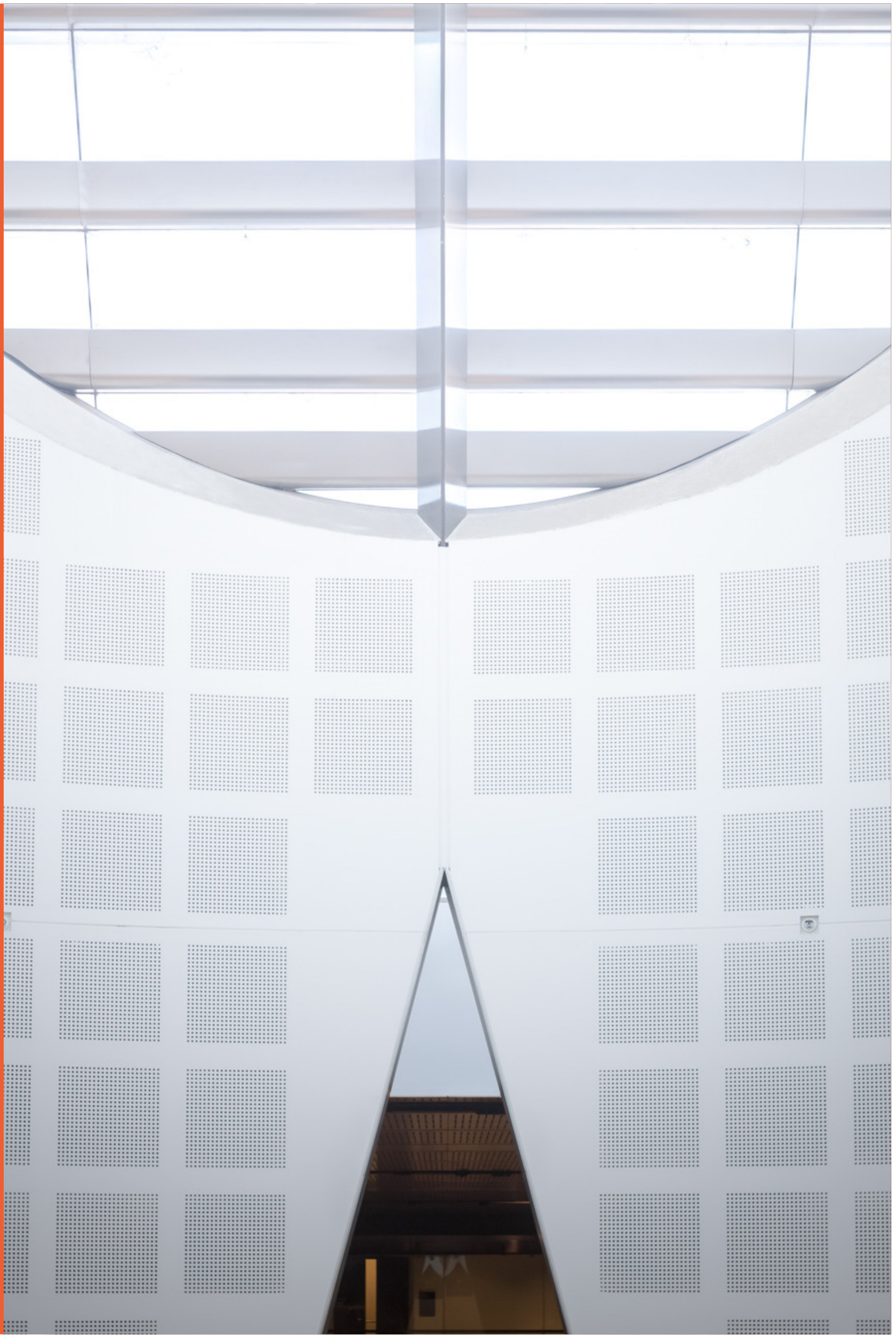
INFO2222
Software security

Presented by

Luke Anderson

THE UNIVERSITY OF SYDNEY



Overview

Today's agenda

- Software and vulnerabilities
  - Buffer overflow
  - Missing input checking
  - Time-of-check to time-of-use
  - Formal defence methods (testing, tools, best practices)
- The online context
- Malicious software (malware)
  - Viruses, worms etc
  - Trapdoors, rootkits, key loggers

Introduction

- If an attacker can run malicious programs on your computer while pretending to be you, it can do serious damage or lead to valuable data being lost or breached.

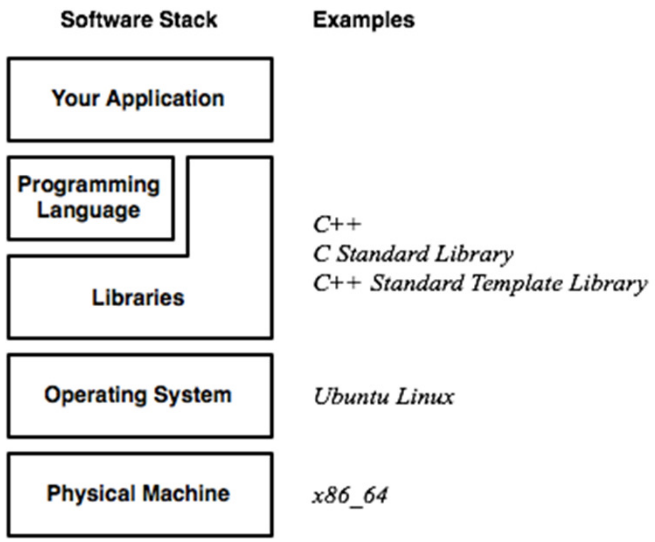
- There are many techniques for introducing malicious code into your machine and getting it to execute.

- This lecture will give an overview of these techniques and how to defend against them.

Software and software vulnerabilities

Building software

- User clicks a button
- Translates to:
  - **Turn** camera **ON**
- Translates to:
  - get the active camera
  - enable capture (**true**)
- Translates to:
  - Locate device handle with name X
  - Set field of structure
  - Flush to device
- …
- Translates to:



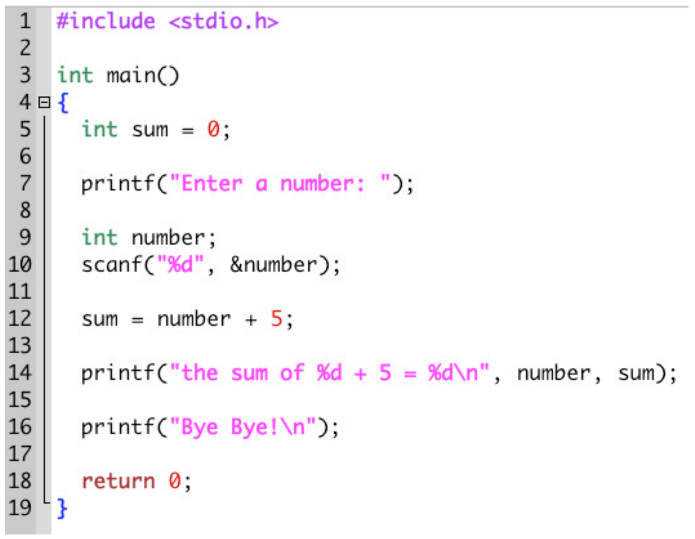
copy N bytes **FROM** Bus 001 Device 005: ID 0624:0249 **TO** address 0x0021

- …

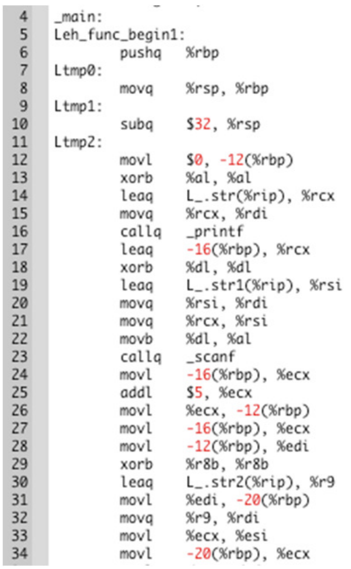
  0010100100100100100100101001110101110001 01…

Building software

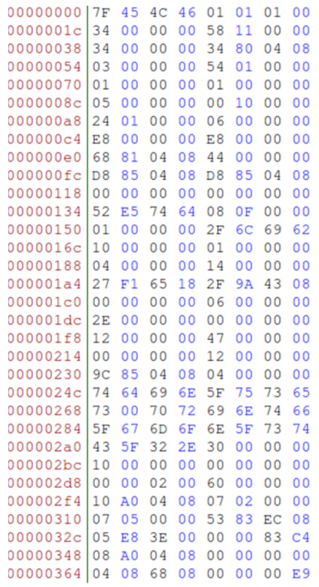
- Humans need *some* language to then be encoded as machine instructions
- A **compiler** translates the language semantics/instructions to machine instructions
  1. High level language -> Low level language -> Machine code
  2. Low level language -> Machine code



```c
#include <stdio.h>

int main()
{
    int sum = 0;

    printf("Enter a number: ");

    int number;
    scanf("%d", &number);

    sum = number + 5;

    printf("the sum of %d + 5 = %d\n", number, sum);

    printf("Bye Bye!\n");

    return 0;
}
```



```
_main:
Leh_func_begin1:
        pushq   %rbp
Ltmp0:
        movq    %rsp, %rbp
Ltmp1:
        subq    $32, %rsp
Ltmp2:
        movl    $0, -12(%rbp)
        xorb    %al, %al
        leaq    L_.str(%rip), %rcx
        movq    %rcx, %rdi
        callq   _printf
        leaq    -16(%rbp), %rcx
        xorb    %dl, %dl
        leaq    L_.str1(%rip), %rsi
        movq    %rsi, %rdi
        movq    %rcx, %rsi
        movb    %dl, %al
        callq   _scanf
        movl    -16(%rbp), %ecx
        addl    $5, %ecx
        movl    %ecx, -12(%rbp)
        movl    -16(%rbp), %ecx
        movl    -12(%rbp), %edi
        xorb    %r8b, %r8b
        leaq    L_.str2(%rip), %r9
        movl    %edi, -20(%rbp)
        movq    %r9, %rdi
        movl    %ecx, %esi
        movl    -20(%rbp), %ecx
```

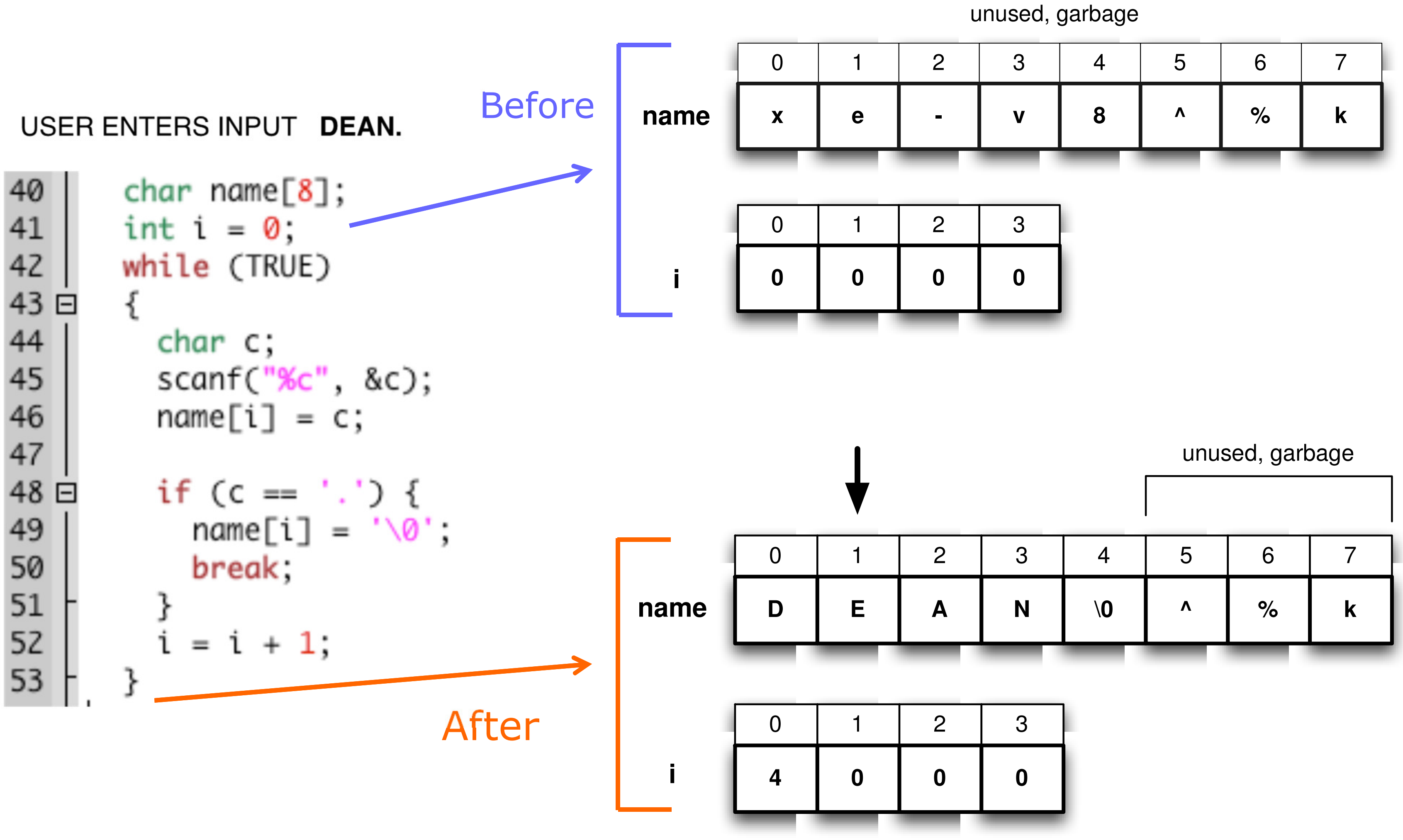


```
00000000|7F 45 4C 46 01 01 01 00
0000001c|34 00 00 00 58 11 00 00
00000038|34 00 00 00 34 80 04 08
00000054|03 00 00 00 54 01 00 00
00000070|01 00 00 00 01 00 00 00
0000008c|05 00 00 00 00 10 00 00
000000a8|24 01 00 00 06 00 00 00
000000c4|E8 00 00 00 E8 00 00 00
000000e0|68 81 04 08 44 00 00 00
000000fc|D8 85 04 08 D8 85 04 08
00000118|00 00 00 00 00 00 00 00
00000134|52 E5 74 64 08 0F 00 00
00000150|01 00 00 00 2F 6C 69 62
0000016c|10 00 00 00 01 00 00 00
00000188|04 00 00 00 14 00 00 00
000001a4|27 F1 65 18 2F 9A 43 08
000001c0|00 00 00 00 06 00 00 00
000001dc|2E 00 00 00 00 00 00 00
000001f8|12 00 00 00 47 00 00 00
00000214|00 00 00 00 12 00 00 00
00000230|9C 85 04 08 04 00 00 00
0000024c|74 64 69 6E 5F 75 73 65
00000268|73 00 70 72 69 6E 74 66
00000284|5F 67 6D 6F 6E 5F 73 74
000002a0|43 5F 32 2E 30 00 00 00
000002bc|10 00 00 00 00 00 00 00
000002d8|00 00 02 00 60 00 00 00
000002f4|10 A0 04 08 07 02 00 00
00000310|07 05 00 00 53 83 EC 08
0000032c|05 E8 3E 00 00 00 83 C4
00000348|08 A0 04 08 00 00 00 00
00000364|04 08 68 08 00 00 00 E9
```

$$\begin{tikzcd}
\text{High level} \arrow[rr] \arrow[dr] & & \text{Machine code} \\
& \text{Low level} \arrow[ur] &
\end{tikzcd}$$

Information Stored In Memory

USER ENTERS INPUT **DEAN.**

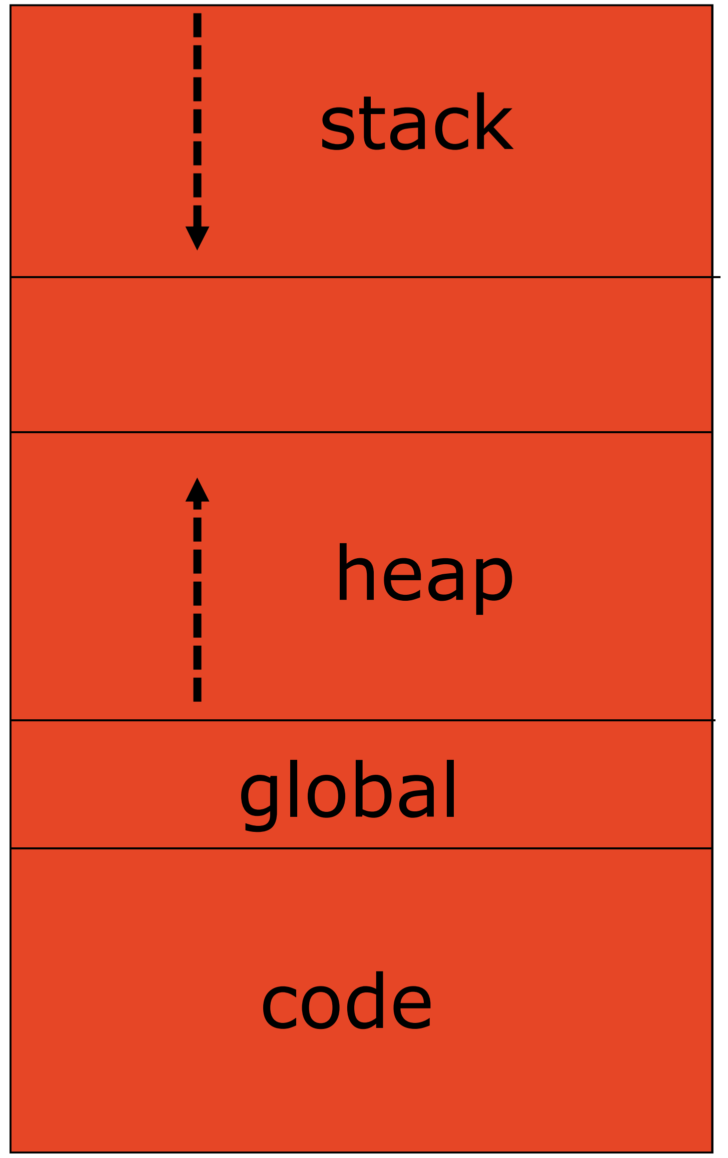


```
char name[8];
int i = 0;
while (TRUE)
{
   char c;
   scanf("%c", &c);
   name[i] = c;

   if (c == '.') {
      name[i] = '\0';
      break;
   }
   i = i + 1;
}
```

Memory

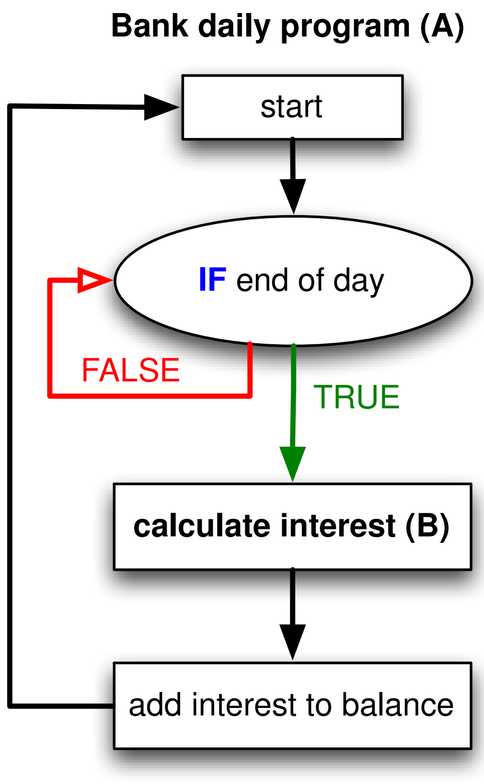
- Typical layout of memory address space

- **Stack:** local variables, function return addresses
- **Heap:** dynamically allocated memory
- **Global:** global variables
- **Code:** program instructions



Example: bank software

Add the interest to an account at the end of the day.



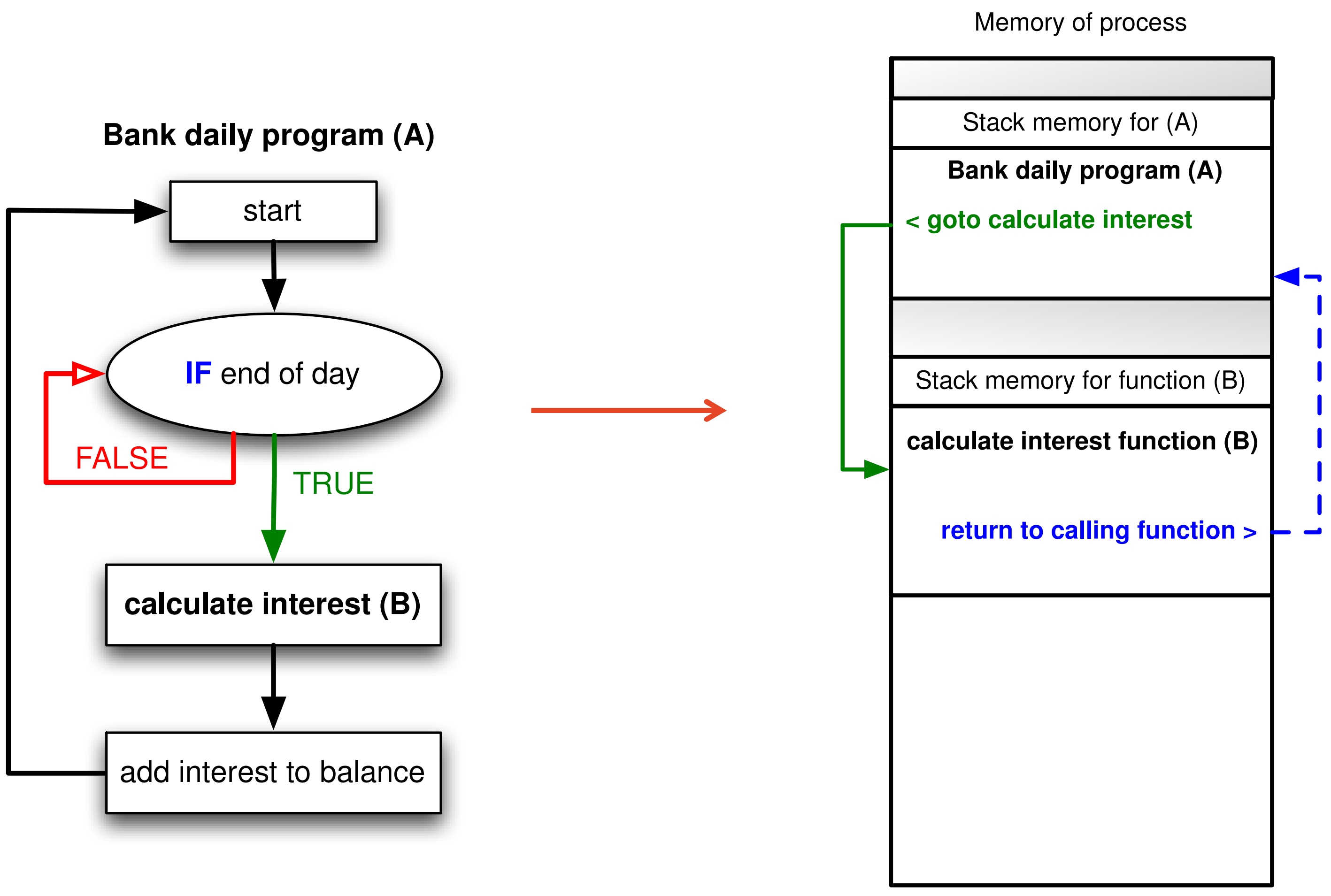
There are numbers to keep track of: time, balances, interest rate etc.

There are some computations that need to happen. These can be compartmentalized into functions.

<- Function A will pause, and call another function B. When B finishes, A resumes.

Stack memory is used to keep track of data related to the function.

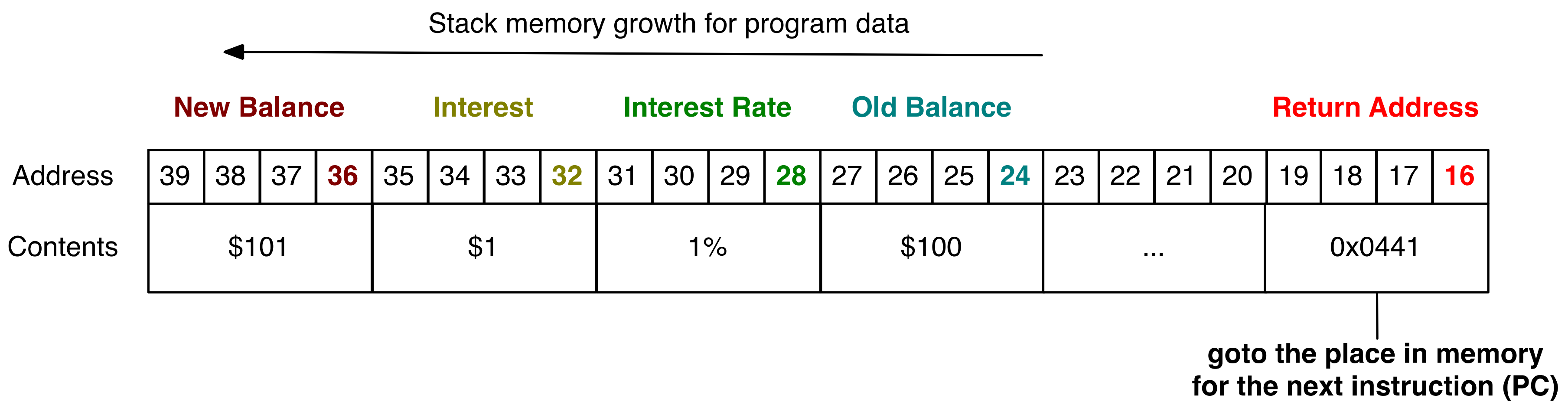
Example: bank software



Layout of Stack Memory

- Stack memory is for housekeeping of function data, including the return address, that is where to go next.

- The value for the return address is the place in code from the **calling** function (for the next CPU instruction).



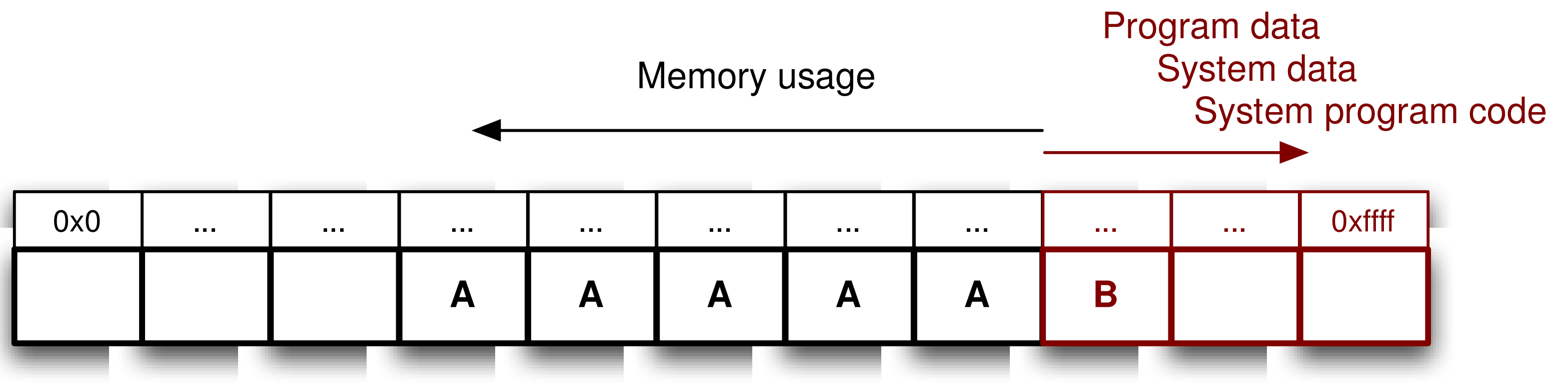
Stack memory growth for program data
New Balance
Interest
Interest Rate
Old Balance
Return Address
Address
Contents
goto the place in memory for the next instruction (PC)

Address	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Contents	$101				$1				1%				$100				...				0x0441			

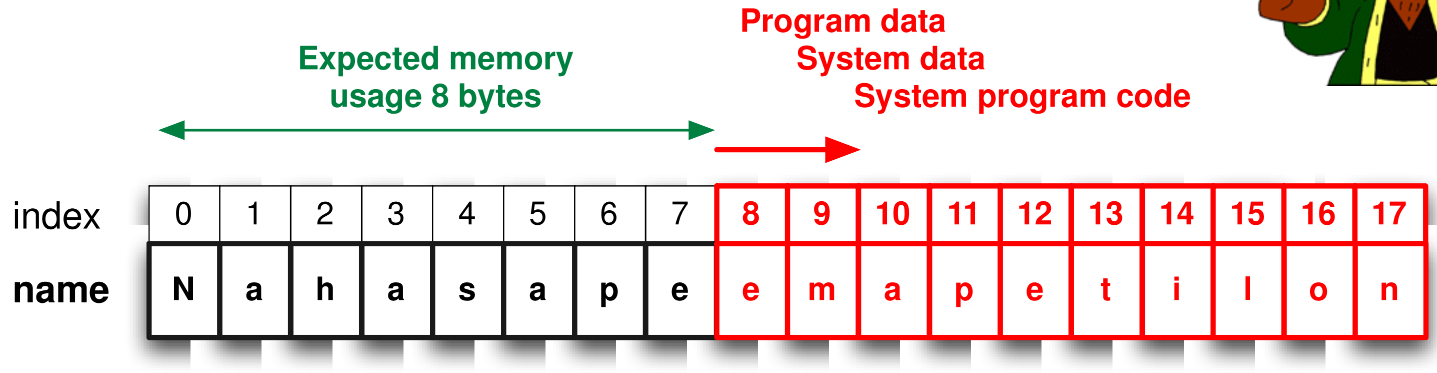
Vulnerability: buffer overflow

- A program defect.
- Normally causes program to fail but may not be triggered by normal input, only by carefully crafted input
  - A major cause of security violations.
- Basic idea:
  - Overrun the part of a data structure with some data that will cause execution of malicious code.
- There are many examples of buffer overflow attacks

Vulnerability: buffer overflow



Apu Nahasapeemapetilon



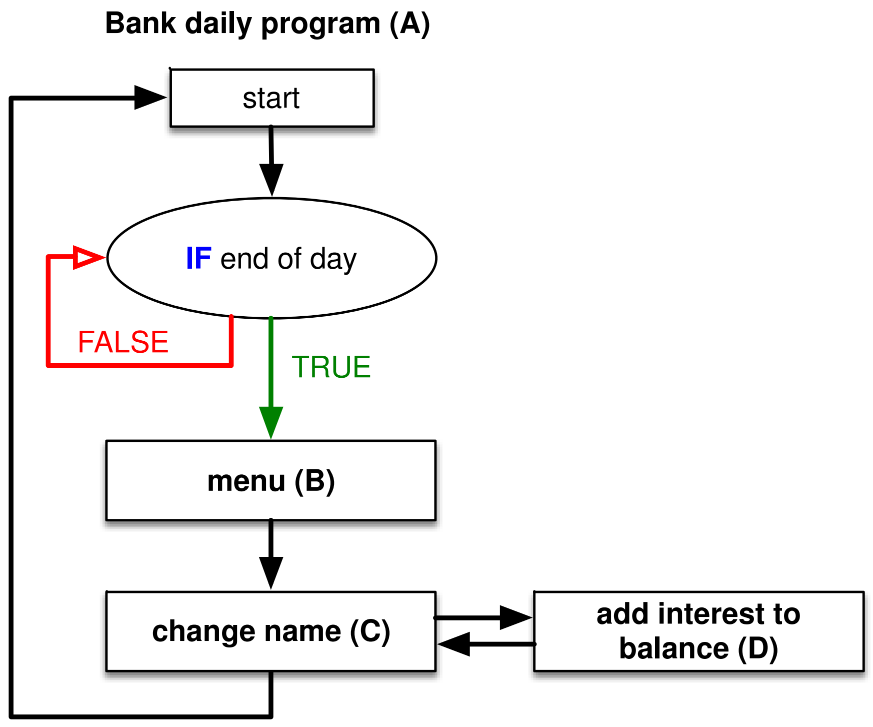
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
name	N	a	h	a	s	a	p	e	e	m	a	p	e	t	i	l	o	n

If this is C, any programmers see what is else missing?

Exploit: bank software

Call function (D) when function (C) completes

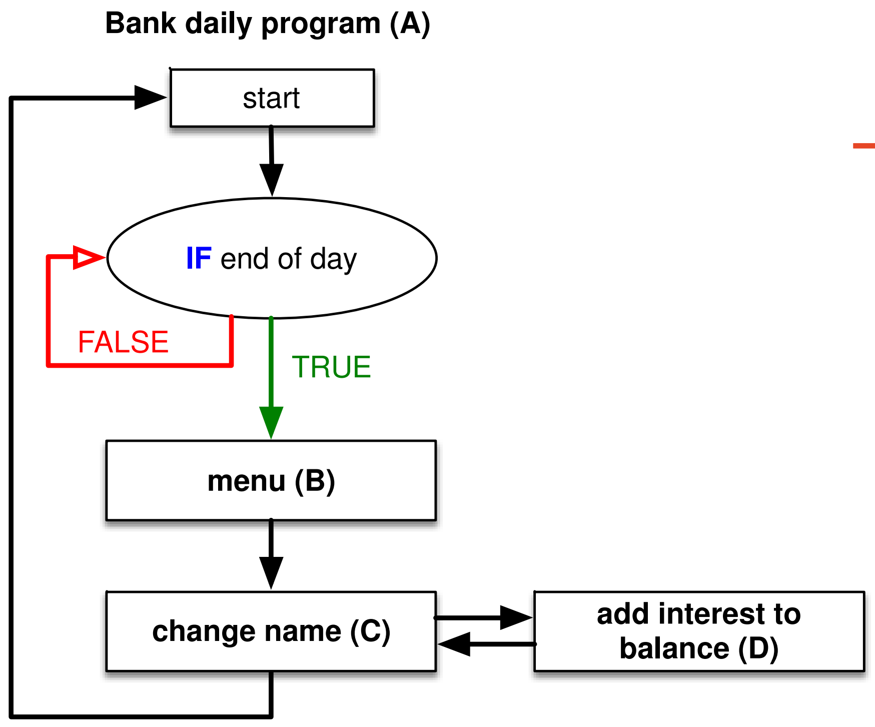
- By changing the Program Counter for the jump instruction memory address
- Still perform (C), but also do something else!

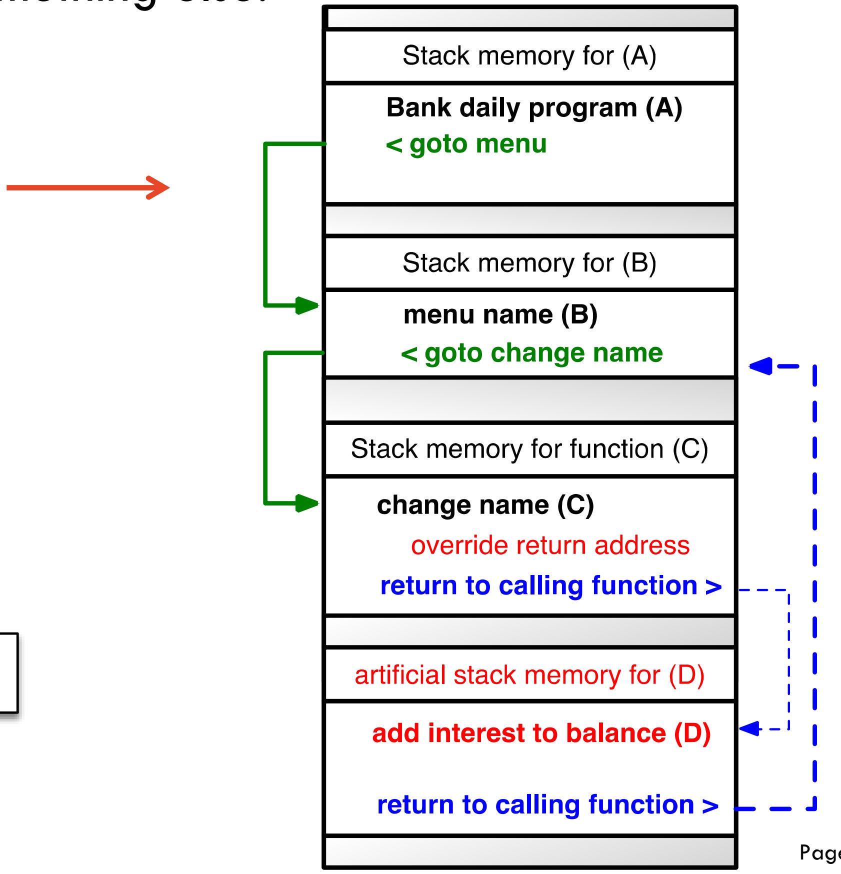


Exploit: bank software

Call function (D) when function (C) completes

- By changing the Program Counter for the jump instruction memory address
- Still perform (C), but also do something else!





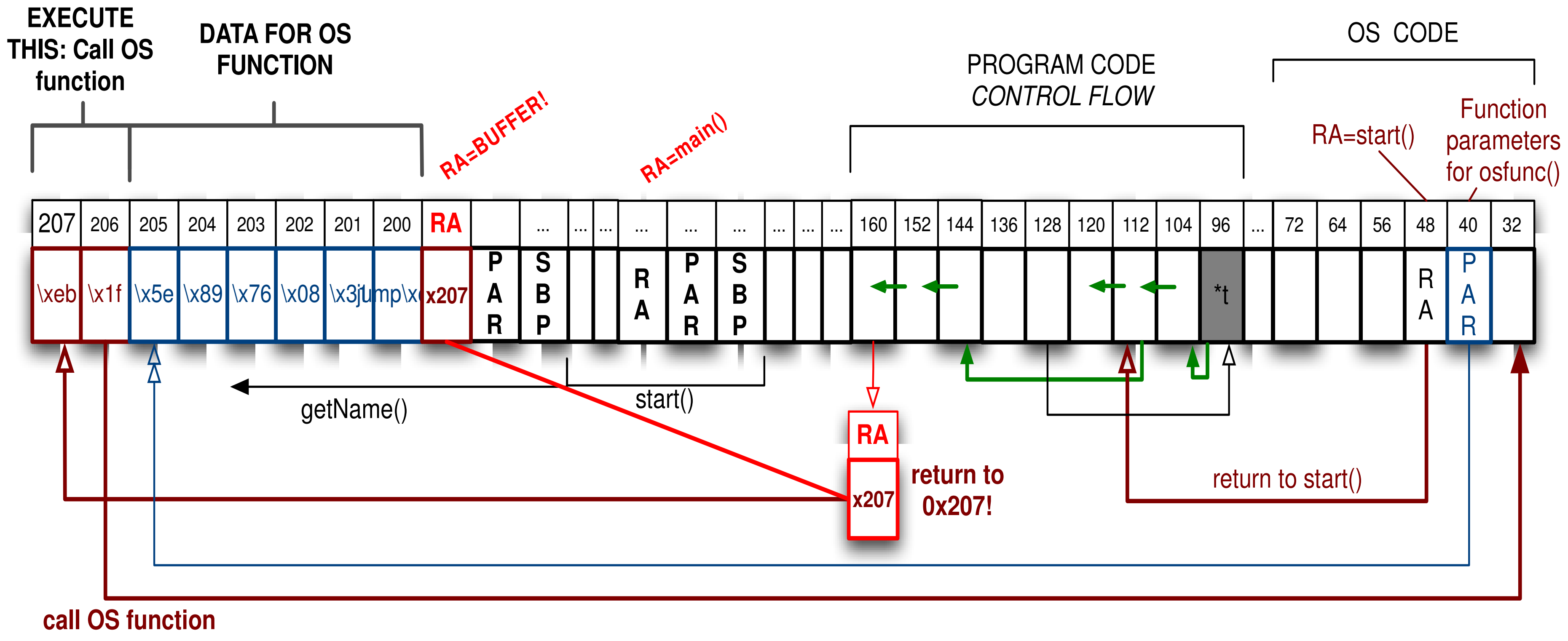
Stack memory for (A)
Bank daily program (A)
< goto menu
Stack memory for (B)
menu name (B)
< goto change name
Stack memory for function (C)
change name (C)
override return address
return to calling function >
artificial stack memory for (D)
add interest to balance (D)
return to calling function >

Steps in buffer overflow exploitation

- **Find** the overflow
  - A discovery process
- **Change the control flow** of the program
  - Override memory locations with instructions
- **Insert** foreign program into memory
  - Retrieve from disk/internet
- **Execute** attacker's program
  - Start doing other things on the machine
- Return to expected code location for normal execution (optional)

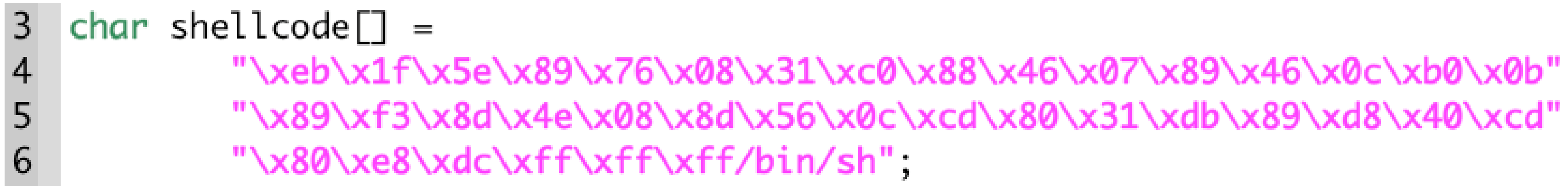
Bypass original program flow

– Arbitrary execution is done via OS function



Inject code as binary data

- Difficult to derive the code as a tiny binary
  - Compression
  - Overwrite stack data may cause program crash



```c
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

- Post production
  - Accessing the binary file also reveals information about the layout
  - Running program has the information stored in the OS memory (hard/soft access)

Issues for the attacker

- Need to introduce binary machine instructions in the string used to overwrite the stack.
- Code has to have absolute address of the entry point, since this overwrites the return address on the stack.
- Need to determine the correct address that buffer will be stored in.
- However once a successful attack vector has been found, it can be re-used to infiltrate any machine that has not had the appropriate control applied.

Issues for the defender

- Understanding how memory is accessed using the functions available to the programmer

- E.g. many C functions have no range checking and so are unsafe
  - strcpy, strcat, gets, scanf

- Extra work because of constant checking
  - validate inputs at every step,
  - sanity check on stored data,
  - must assume that all data handled at any point in the program is of the wrong size or format

Vulnerability: heap overflow

- The heap is used for dynamic data structures.
  - an area of memory is allocated and a **pointer** returned to a program
  - pointer is **data structure** that can be followed ("dereferenced") and that gives you the actual data it is pointing to
  - no automatic checking that the program stays within the allocated area

- Heap overflow has similar characteristics as stack buffer overflow.

- May not involve executing malicious code: could overwrite file names causing critical files to be overwritten.

Other vulnerabilities

- Huge category of incomplete checking of data inputs
- Classics, but **very much still relevant today**

- Format string vulnerability:
  - Functions like printf() take parameters that describe how output should be formatted.
  - Can be exploited by submitting parameters as "input"
- URL handling
  - Malformed URLs can cause buffer overflows.
- SQL injection
  - Incomplete checking of DB query parameters.
  - Allows dumping information from the database in some cases.

Time of check v. time of use

- A program may check the status of a file and then open it

- If the attacker can (from outside) change the status between these checks a program might be tricked into opening the wrong file

E.g. disallow the opening of symbolic links by first checking if the name is a symbolic link and if not, opening the file

  - Attacker may be able to change the name to a symlink after the check

Access control matrix

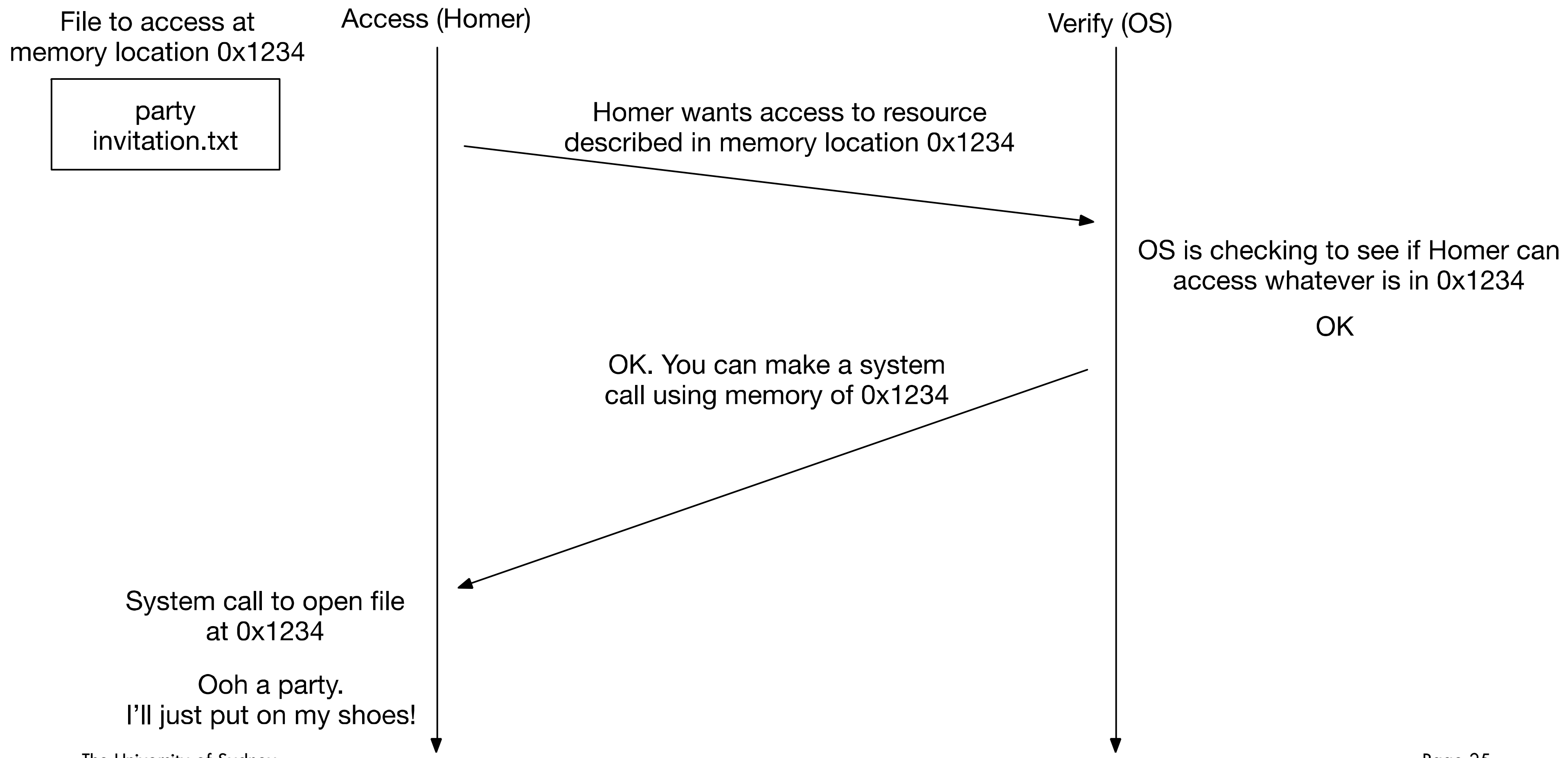
	party invitation.txt	ned flanders bank details.txt
homer	YES	NO
ned flanders	YES	YES

Time of check vs. time of use

Access control matrix

	party invitation.txt	ned flanders bank details.txt
homer	YES	NO
ned flanders	YES	YES

Normal access



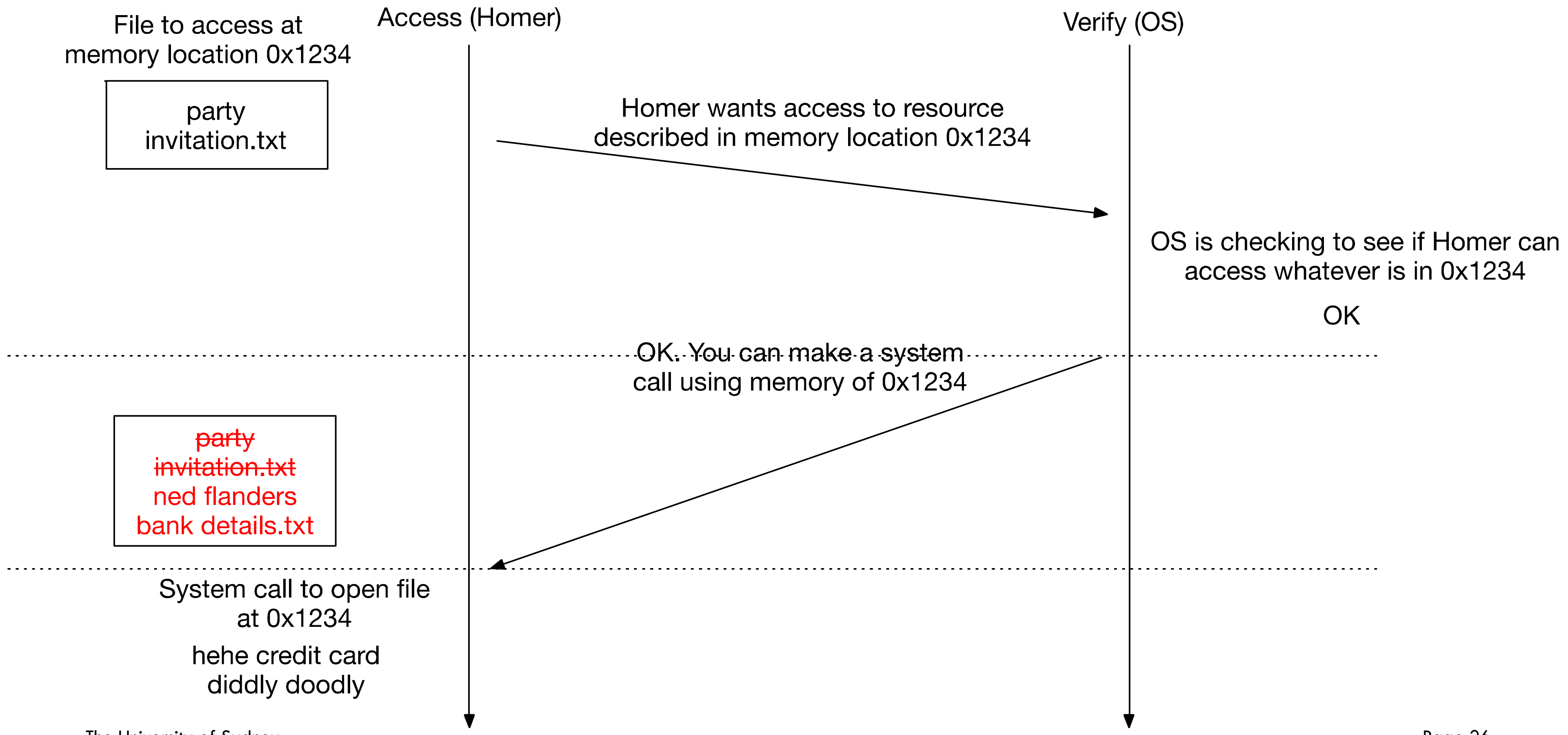
File to access at memory location 0x1234
party invitation.txt
Access (Homer)
Verify (OS)
Homer wants access to resource described in memory location 0x1234
OS is checking to see if Homer can access whatever is in 0x1234
OK
OK. You can make a system call using memory of 0x1234
System call to open file at 0x1234
Ooh a party. I'll just put on my shoes!

Time of check vs. time of use

Access control matrix

	party invitation.txt	ned flanders bank details.txt
homer	YES	NO
ned flanders	YES	YES

Race condition exploit



File to access at memory location 0x1234
party invitation.txt
Access (Homer)
Verify (OS)
Homer wants access to resource described in memory location 0x1234
OS is checking to see if Homer can access whatever is in 0x1234
OK
OK. You can make a system call using memory of 0x1234
~~party invitation.txt~~ ned flanders bank details.txt
System call to open file at 0x1234
hehe credit card diddly doodly

Defences: buffer overflow

- Modern programming language have proper boundary checking: C#, Java, Python, Go, Rust, …
  - C/C++ are commonly used for performance/embedded systems/IoT
  - Note that Go and Rust were expressively developed as new systems languages with similar performance
  - Much legacy code exists: the problem will not go away by itself
- Some compilers can warn of problems
- In C/C++, use the **safe versions** of certain functions
  - Details beyond this unit; but when learning C, make sure you learn about the safe versions **from the start**
  - "l-versions" are safest: strlcpy(), strlcat(), ... – not available on all systems. Truncate automatically and add terminator character.
  - "n-versions" are next best: strncpy(), strncat(), ... – truncate, but do not add termination character (can be a trap for your program)

Defences: buffer overflow

- Static code analysis tools
  - Check program for errors without executing it
  - Many approaches: formal methods; abstract interpretation; symbolic execution; data-flow analysis; simpler (style) checks
  - Some built into compiler, e.g. clang for C
  - Some common names: cppcheck, lint, splint, ...

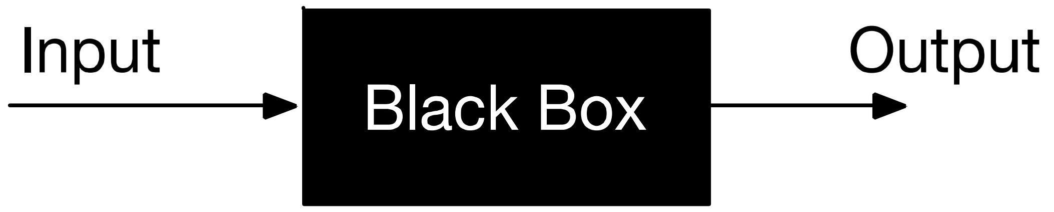
- Dynamic code analysis
  - Run time checking tools: valgrind, purify,….
  - Check for memory corruption, detailed analysis of machine instructions
  - Detailed profiling: also useful to find execution bottlenecks
  - Programs run much more slowly as many extra instructions are added

Defences: unit tests

- Core idea is to break program code into small units that can be tested separately
- Then write dedicated test programs that run the units with all possible inputs that you can think of
- Very useful when changes are made to units – when unit tests suddenly break, you know you have an error
- Testing is very important for all program development but can only show the presence, not the absence of bugs.
- Even carefully designed tests will often fail to find the "unusual" bugs.
- Many other forms of testing, e.g. integration tests
- Topic of Software Engineering

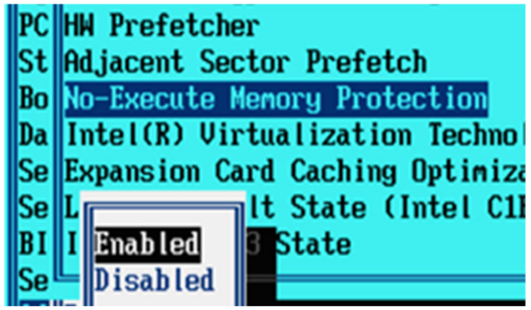
White box vs. black box testing

- Unit tests are white box tests:
  - You know the code that you are testing
  - You can write dedicated tests to check all code paths
- Black box testing means to test code where you do not have the source code
  - Much reduced possibilities, but very important when you need to use someone else's binary



Defence: No-execute memory areas

- Use the hardware to prevent execution of instructions anywhere but in the code area of memory
- Disallows the attacker to put their data into some area and point the return pointer to it
- All modern CPU architectures support it (even ARM)
- Still allows the overwriting of return addresses: make the program return to the wrong place
  - Can be exploited – "return-oriented programming", where you piece a malicious code sequence together from the real source code



```
PC HW Prefetcher
St Adjacent Sector Prefetch
Bo No-Execute Memory Protection
Da Intel(R) Virtualization Techno
Se Expansion Card Caching Optimiz
Se L           lt State (Intel C1
BI I  Enabled    State
Se    Disabled
```

Defence: ASLR, Stack Canaries, Shadow Stacks

- **Address Space Layout Randomization (ASLR):**
  - Randomizes how memory is organized – addresses become unpredictable
  - Attacker can't predict where functions are located in memory
  - Modern Operating Systems do this

- **Stack Canaries:**
  - Intentionally placing values on the stack
  - Periodically checking they haven't been touched. If they have, abort.

- **Shadow Stacks:**
  - Keep a second copy of the stack elsewhere for critical parts of the application.
  - Refer to it to check for changes.

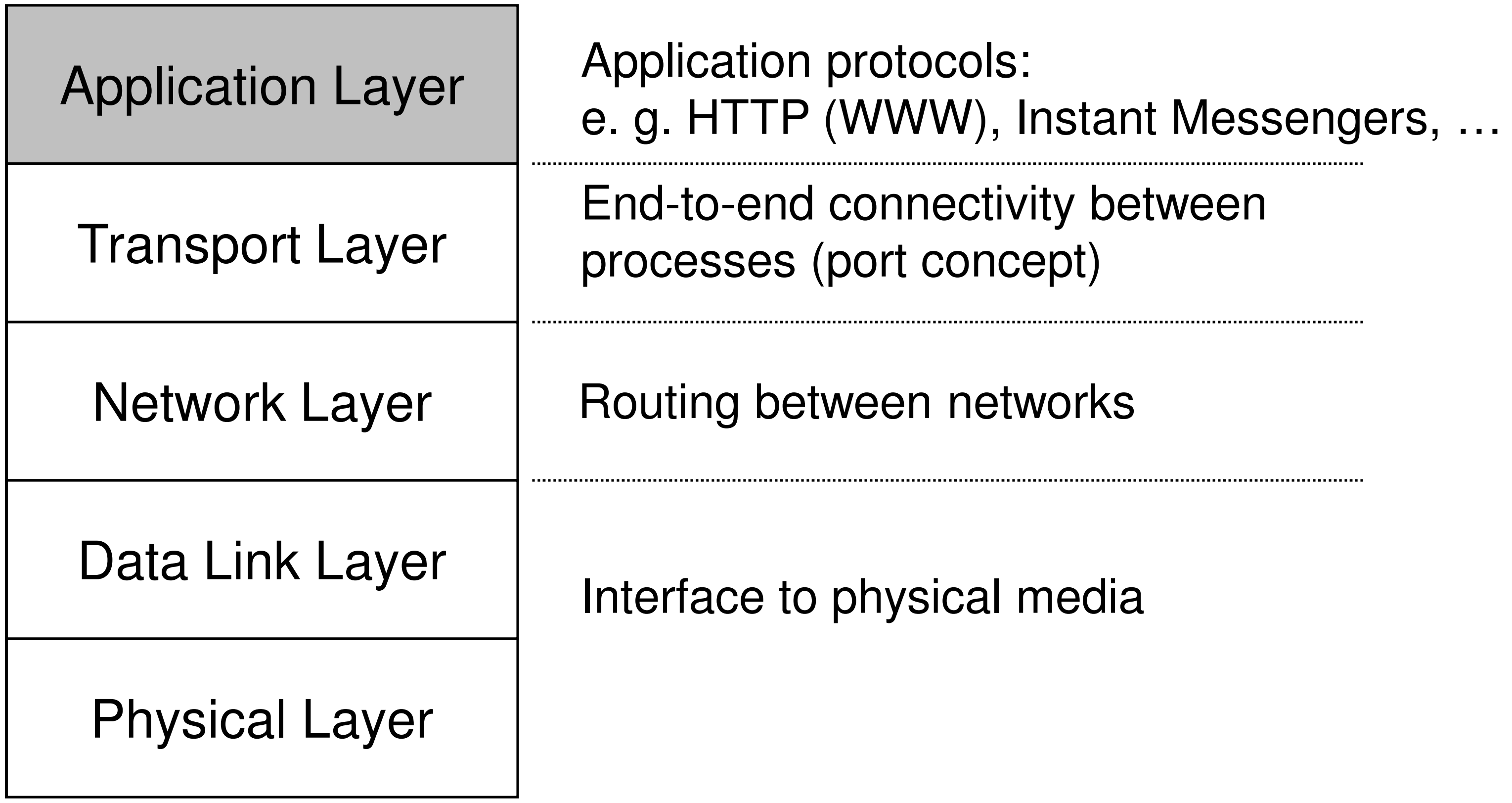
Defence: fuzzing

- Input to the program is **randomly** generated and large amounts of test data are used.
  - program failures indicate a bug
  - the bug might be used by an attacker!
  - very effective testing technique
  - part of standard testing for many products
- Standard tools exist (e.g. american fuzzy loop)
- Randomly generate interesting input values/strings that are outside known boundaries
  - eg pass a URL to a web browser that is 10,000 characters long
  - eg ask a database to delete the next -10 items
  - eg feed text that is in a non standard character set

Software operating online

- Our previous examples had not distinguished where malicious input is coming from
- A buffer overflow, for example, can be triggered by a local user or program against another program that is taking input via console, pipes, or local inter-process communication
- Much of our software today, however, operates online:
  - Exposed to data being sent from many possible origins
  - The attacks you have learnt about are still very relevant – but they are not the dominant ones anymore
- Today, the WWW is the most popular use of the Internet
  - Many people don't even know there is a difference
  - WWW software takes user input via a multitude of ways!
  - We are now going to look at a few of them

Recap: Internet Protocol Suite


Application Layer
Transport Layer
Network Layer
Data Link Layer
Physical Layer
Application protocols:
e. g. HTTP (WWW), Instant Messengers, …
End-to-end connectivity between processes (port concept)
Routing between networks
Interface to physical media

Most important Web technologies

Name	Used for	Comment
HTML	Document structure and content	Should not describe how document should be rendered; but can be used for that
CSS	Document rendering	Intended to describe how a document should be rendered
HTTP	Carrier protocol	Browsers and web servers communicate via HTTP
Cookies	Session state keeping	Needed to make the WWW useful
URL/URI	Document location	Describe where document can be found
JavaScript	Client-side computation and interaction	Allows dynamic web sites
Flash	Client-side code execution	Dying out – used to make dynamic sites, but propietary and poor security record

Introduction to the World Wide Web

- Conceived in 1989/90 by Tim Berners-Lee at CERN
- Hypermedia-based extension to the Internet on the Application Layer
  - Any information (chunk) or data item can be referenced by a Uniform Resource Identifier (URI)
  - URI syntax (defined in RFCs) :
    `<scheme>://<authority><path>?<query>#<fragment>`
  - Special case: URL ("Locator")
    `http://www.net.in.tum.de/de/startseite/`

- Currently, most vulnerabilities are found in Web applications

HTML and Content Generation

- HTML is the *lingua franca* of the Web
  - Content representation: structured hypertext documents
  - HTML documents – i. e. Web pages – may include:
    - JavaScript: script that is executed in browser
    - Dying out: Java Applets (Java programs)
    - Dying out: Flash
- Much content is created dynamically by the server
- Dynamic Web pages interact with the user
- Examples of server-side technology/languages:
  - PHP, Python, Perl, Ruby, …
  - Java (several technologies), ASP.NET
  - Possible, but rare: C++ based programs

HTTP

- HTTP is the carrier protocol for HTML
  - State-less: server does not keep state information about client
  - Mostly simple `GET/POST` semantics (`PUT` is possible)
- Today: real work flows implemented with HTTP/HTML
  → need to keep state between different pages
  → **sessions**

Sessions Over HTTP

- Sessions: many work-arounds around the state-less property
  - Cookies (later)
  - Session-IDs (passed in HTTP header)
  - Parameters in URL
  - Hidden variables in input forms (HTML-only solution)

- Session information is a valuable target
  - E. g., online banking: credit card or account information

- Session IDs in the URL can also be a weakness
  - Can be guessed or involuntarily compromised (e. g. sending a link)
    → "session hijacking"

Cookies

- Small text files that the server asks the browser store
  - Client authenticates to server, receives cookie with a secret value
  - Uses this value to keep the session alive when transmitting HTTP(s)

- Problematic: which cookies is a site allowed to access?
  - `abc.com` must not access cookies for `xyz.com`

- Cookies come with a security policy implemented in the browser

JavaScript

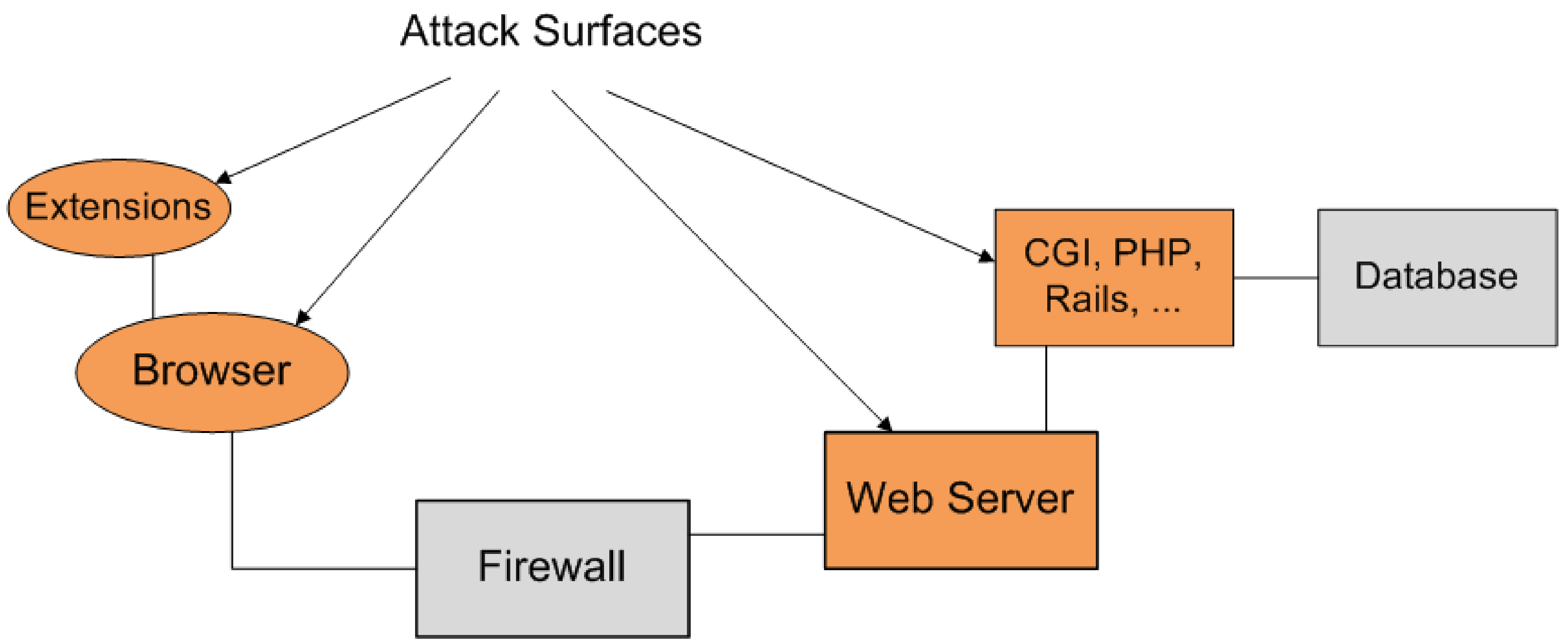
- Script language that is executed on client-side
  - Object-oriented with C-like syntax, but multi-paradigm
  - Allows a Web site to execute programs in the browser
- The Web is less attractive without JavaScript – but anything that is downloaded and executed by a client may be a security risk
- Security Issues:
  - Allows authors to write malicious code
  - Allows cross-site attacks
- Defences:
  - Sandboxing of JavaScript execution
  - Same-origin policy (SOP)

Same-Origin Policy (SOP) and CORS

- Due to the nature of the web, resources can be downloaded from many places ("origins")
  - But this would allow a JavaScript from some resource to interact with a JavaScript from another – undesired because no control over external resources
  - Original idea (Netscape, 1995!): two JavaScript contexts are allowed access to each other if and only if (protocol, host, port) associated with them match **exactly**
- Today: more fine-grained control
  - Using the Cross-Origin Resource Sharing (CORS) mechanism, sites can defines from which other sites they load resources
  - Allows to define which origins are considered trusted
  - Puts the onus of correct configuration and maintenance on the web operator – defining a correct and useful CORS can be difficult

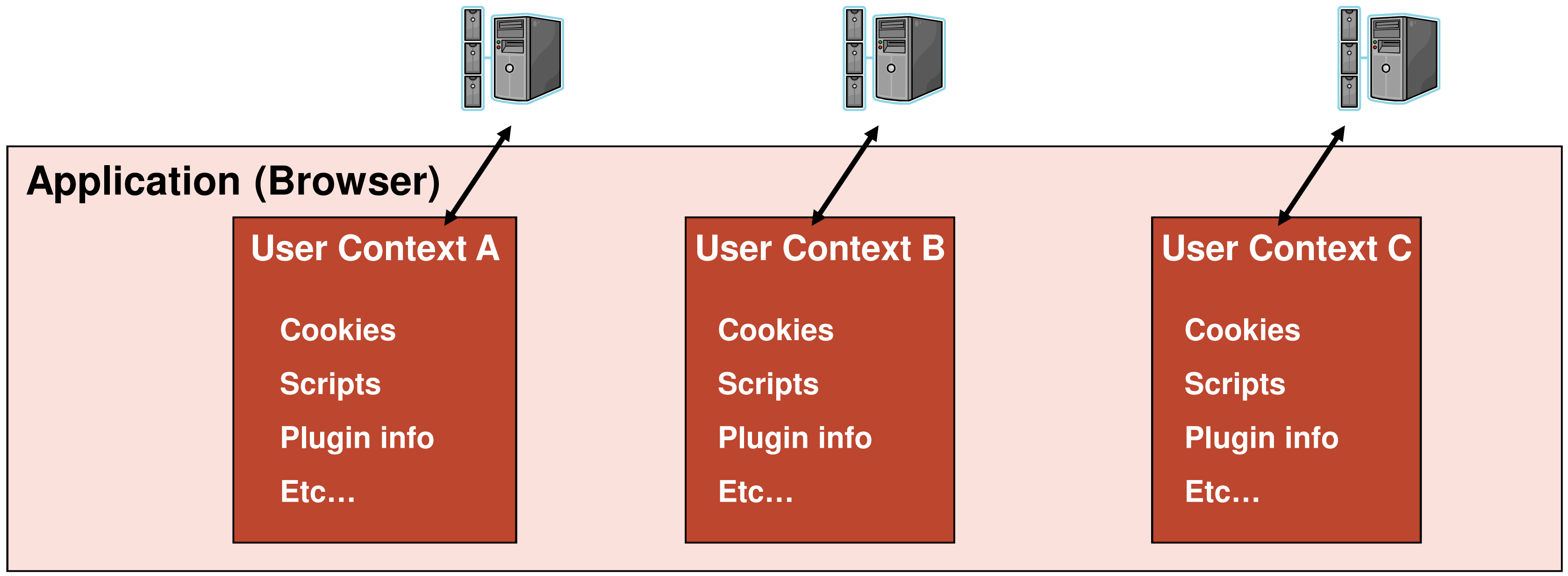
One Step Back: why is the WWW so vulnerable?

- Many important business transactions take place
- Much functionality, much complexity in software
  → many attack vectors, huge attack surface
- Any web application that interacts with the outside world must be open by definition and reachable even across a firewall



Informal definition: contexts

- Context (in general): collection of information that belongs to a particular session or process
  - Useful abstraction that helps us to classify the target of an attack
  - Here: not a formal definition, nor a model of actual implementation
- User Context (in a browser):
  - Collection of all information that “belongs” to a given session
  - Cookies, session state variables, plugin-specific information…
  - JavaScripts: downloaded and executed → obey same-origin policy and CORS!
  - Information from session A should not be accessible from Session B
  - Client and server must remain synchronized w.r.t. state information



Attack 1: Session Variables

- **Target of attack:**
  Synchronization of state information between client and server (session management is attacked)
- **Typical scenario:**
  Exchange between client and server that takes
  several steps to complete - such as purchases
- **Typical approach of attack:**
  Swap state information in one step
- **Cause of vulnerability:**
  Server (or client) relies on information sent by the other party instead of storing it itself

Attack 1: dangerous workflow

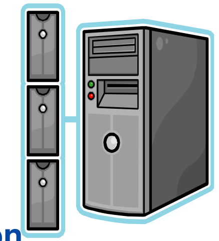
State:

A→S

"toy for $10"

Browser = client

Seller = server



State:

A→S

"$10 for the toy"

A: Put "toy for $1" into basket

2) Background: Credit card verification

Has credit!

S: Offer checkout of "toy for $10"

Offer for "toy for $10"

A: Acknowledge purchase: "toy for $10"

(pay...)

S: "Thanks, shipping to you now"

Receives toy

Question: where do you keep the session information "toy for $10"?

If your answer is "in the cookie": serious mistake.

In fact, S must not trust any information by the browser. We show you why now.

Attack 1: against synchronization of state information

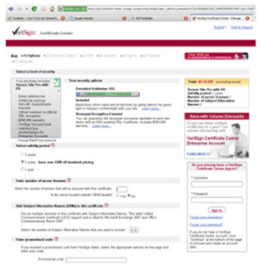
In this example, all state information is stored on client-side and retransmitted in each step (e. g. by reading from a cookie). The server does not store state.

Browser = client

State:

A→S

"toy for $10"



Seller = server

State:

A→S

"toy for $10"

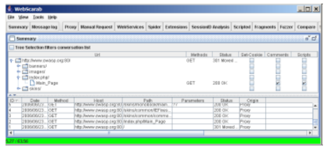
A: Put "toy for $1" into basket

2) Background:
Credit card verification

A has credit

CA: Offer checkout for "toy for $10"

Offer for "toy for $10"



Swap variables on the fly: "toy → Porsche"

A: Acknowledge purchase: "Porsche for $10"

(pays...)

S: "Thanks, shipping to you now"

Has Porsche

Why was the attack possible?

- In our example, all state information was kept on client-side in a cookie
- All the attacker did was to swap "toy" for "Porsche" in the second HTTP request
- The server allowed the purchase for the wrong item because it failed to notice that the *item in the first request was not the same as the item in the second request.*
- That was possible because the relevant information was not stored on server-side

- Do you think this is too easy and will not happen “in the real world”?
  - In fact, this used to be one of the most common mistakes made in custom-built shopping portals
  - We would wager a bet that a large number of untrained developers will get it wrong the first time

Defence and mitigation

- The example allows us to phrase two important guidelines that are **always valid in the online world**
- **For each entity in the protocol:**
  - Everything **relevant** for the **correct outcome** must be stored **locally**
  - Note: it can be difficult to identify this information if you have complex work-flows…
- **All Input Is Evil**
  - Always treat all input as untrusted
  - Never use it without verification
  - Note: what if the server uses Javascript to "force" browser to behave correctly? → just deactivate JavaScript → client still in control!

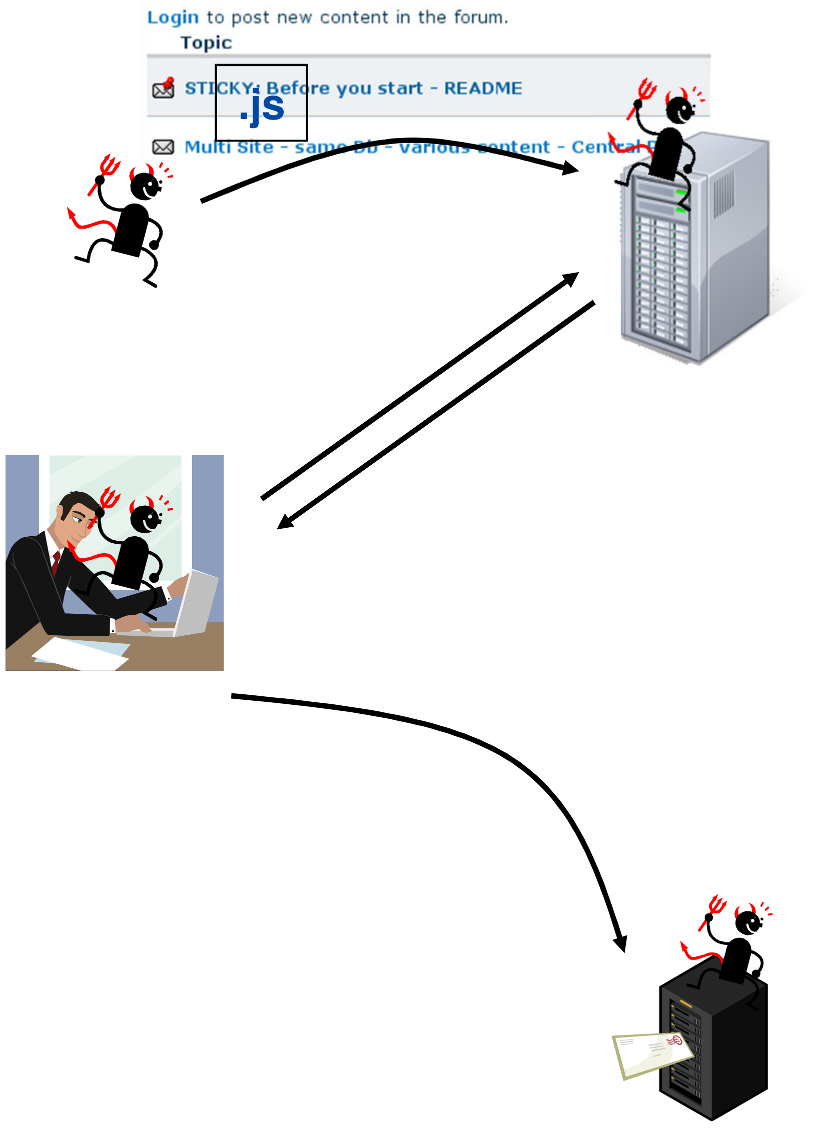
- But is this all you need to know? No! It's the first, smallst step!

Cross-Site Scripting (XSS)

- **Target of attack:**
  - Attempt to access user context from outside the session
  - Goal is to obtain confidential information from the user context
- **Typical scenario:**
  - User surfing the Web and accessing a Web site with JavaScript
- **Typical approach to attack:**
  - Attacker plants a malicious script on a Web page
  - the script is then executed by the user's browser
- **Cause of vulnerability:** two-fold
  - Attacker is able to plant malicious script on a Web page
    $\rightarrow$ vulnerability in Web software needed
  - User browser executes script from a Web page
    $\rightarrow$ user's "trust" in Web site is exploited
- XSS is still one of the most common attacks today!

Cross-Site Scripting: typical attack

- Stage 1: Attacker injects malicious script
  - Here: in a Web forum where you can post messages
  - In addition to normal text, the attacker writes: `<script>[malicious function]</script>`
  - The server accepts and stores this input
- Stage 2: Unaware user accesses web forum
  - Here: reads poisoned message from attacker
  - User receives:
    ```
    <p>Hello, this is a harmless message
    <script>[malicious function]</script>
    </p>
    ```
  - Everything within `<script>` is executed by browser *in the user's context*
- Possible consequences:
  - Script reads information from cookies etc. and sends it to attacker's server
  - Script redirects to other site → download trojan etc.


Login to post new content in the forum.
Topic
STICKY: Before you start - README
.js
Multi Site - same DB - various content - Central

Cross-Site Scripting: why does it work?

- Reason 1: The Web application did not **sanitize** input it received
  - Remember: **all input is evil**; and the attacker can **choose** their input
  - If the web app had just dropped all HTML input, there would be no script uploaded → and none executed in the browser
  - Unfortunately, many Web sites allow users to post at least some HTML
    → a nice feature, but dangerous

- Reason 2:
  The user had trusted the Web site and did not assume
  malicious content could be downloaded and executed
  → **abuse of trust**

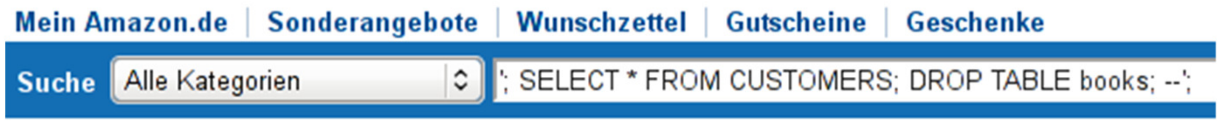
SQL injection

- **Target of attack:**
  - Server context
- **Typical scenario:**
  - Web server runs with an SQL database in the background
  - Attacker wants to extract or inject information to/from the database
- **Typical approach to attack:**
  - Attacker writes SQL code into an input form
  - which is then passed to the SQL database
  - evaluated and output returned
- **Cause of vulnerability:**
  Web server does not sanitize the input and accepts SQL code

- SQL Injection is another classic attack – and still very common

SQL Injection

– Attacker injects SQL into search form:



'; SELECT * FROM CUSTOMERS; DROP TABLE books; --';

– The author of the Web page may have intended to execute:

```
SELECT author,book FROM books WHERE book = '$title';
```

– Through the SQL injection, this has become something like:

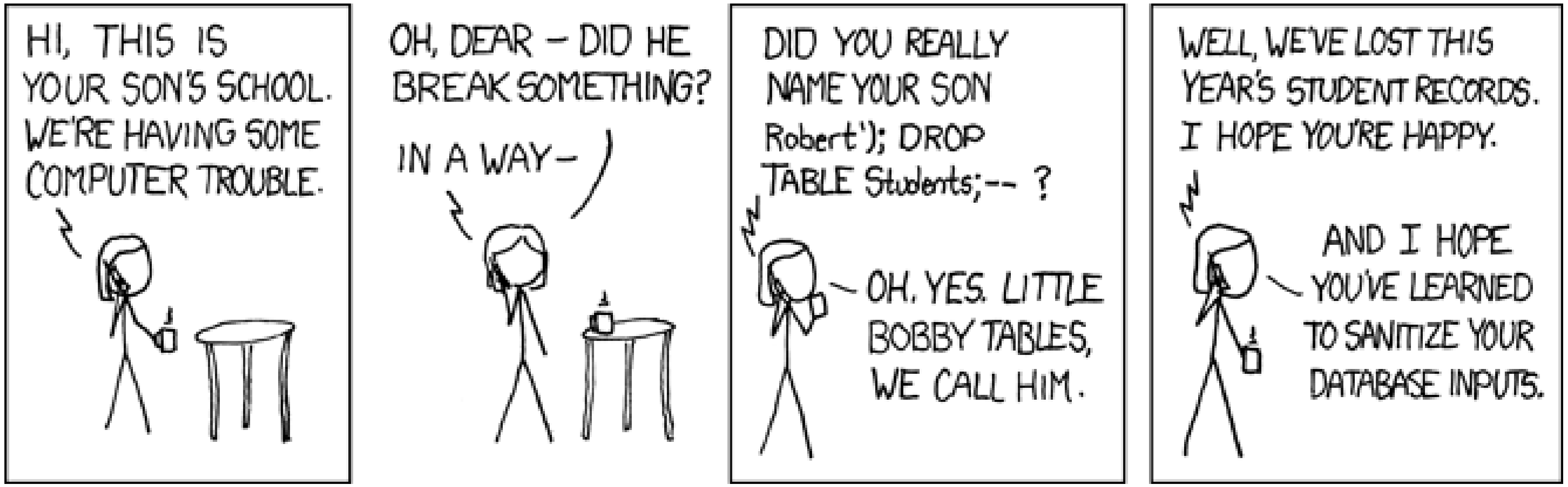
```
SELECT author,book FROM books
WHERE book = ''; SELECT * FROM CUSTOMERS; DROP TABLE
books;
```

– You just lost your catalogue and compromised your customers data

– Amazon, of course, is too clever not too sanitize their input – but it is amazing how many Web sites fail to do so!

– Fortunately, this exact example won't work anymore (only one SQL command allowed by the DB these days)

Sanitize or be sorry


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY –
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

General defences for XSS and SQL Injection

- Paranoid people turn off JavaScript or restrict it to very few sites – works well, but no good web experience
- Better protection can be achieved on **server-side:**
  - Treat all input as **untrusted**
  - **Sanitize** your input and output: proper **escaping**
    - Escape (certain) HTML tags and JavaScript
    - Exceedingly difficult and complex task!
    - Whitelisting is better than blacklisting – the black list may grow
- **Do not write your own escaping routines**
  - Modern script languages offer this functionality by default
- **Use the standard libraries to communicate with the DB**
  - Many eliminate the SQL injection problems for you

Malware

Definition of malware

- Malware: malicious software
  - Usually written with a hostile intention, or at least the intention to disturb the target system (varying degrees of hostility)
  - Needs to be deployed somehow on the target system

- Malware is old:
  - First computer virus dates to mid-1980s
  - Self-replicating, annoying software already existed on the ARPANET (Internet forerunner) in the 1970s
  - New, explosive infection ways via Internet were trialled in the 1990s
  - With commercial importance of digital systems and Internet, we have seen an explosive growth of malware
  - Also allegedly used in (clandestine) warfare today

Distinguishing by propagation type

- Virus
  - Name comes from biology
  - Attaches itself to programs or data files, gets loaded into memory
  - Propagates copies of itself (self-replication), tries to infect other files
  - Modern forms obfuscated themselves: avoid detection
- Worm
  - Spreads stand-alone copies of itself via a network
- Targeted malware: Trojans, logic bombs, trapdoors

Virus infection

The absolute classic malware is the virus

- Just like a biological virus, it cannot execute by itself.
- Appends itself to another program.
  - Virus executes first, does its job, then normal program executes

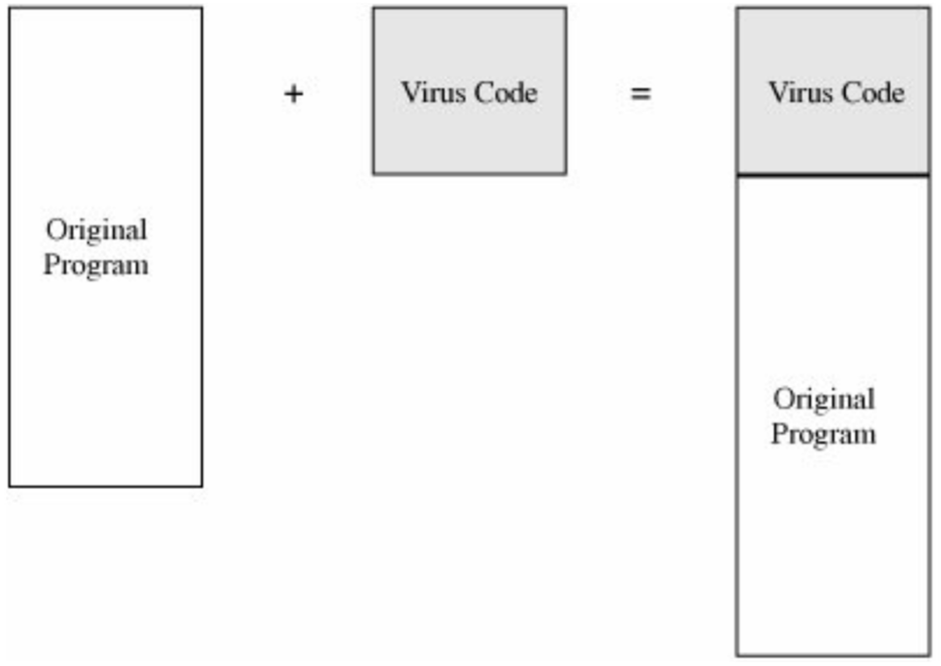
- Can also append to a document.
  - Many docs contain executable commands, eg macros
  - This is still a threat today!

- Replace a program file on disc.
- Boot sector: replaces sectors of OS bootstrap on disk

Virus Infection

- Virus Process
  - Find a host program on the system
  - Find all the function address locations of the program
  - Perform address translation so that they are offset making room for the virus

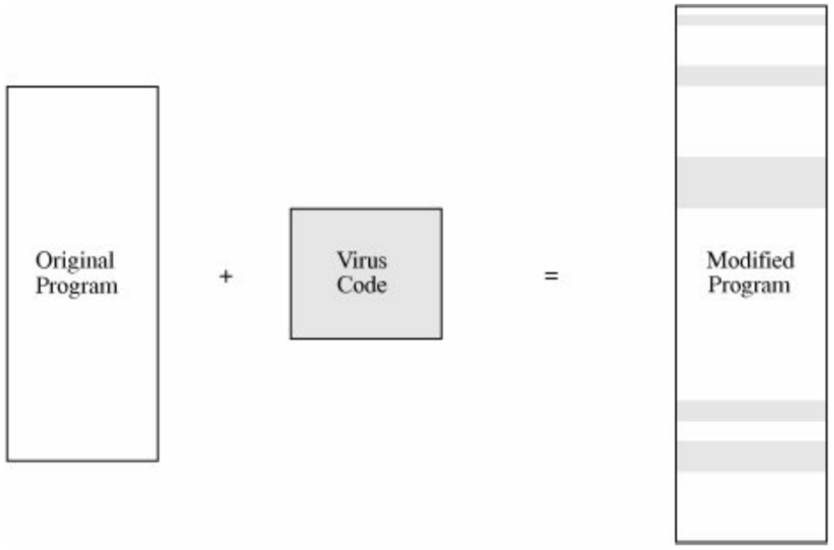
Figure 3-4. Virus Appended to a Program.



Virus Infection

- Trickier methods for hiding in the program.
  - Escape identification of binary signature by distributing self within the program

Figure 3-6. Virus Integrated into a Program.



Virus Infection

- **Evading** antivirus tools: arms race
  - Consistency check of each program/file length
  - CRC32/parity
  - Self encryption
  - Stealth operation
  - Resident in running memory (live)

- **Automatic execution** is the virus golden ticket
  - Document macro
  - Removable media/cdrom/floppy
  - Email attachment
  - Internet: click to download

Defence: virus detection

- Some very simple defences raise the bar considerably:
  - All modern Operating Systems distinguish between admin and user accounts → never run a program as admin/root unless really necessary. Prevents infection of operating system!
  - Don't click on email attachments if the sender is unknown to you
- Antivirus tools (AV) can detect viruses (and other malware)
  - Generally based on signatures
    - Known binary strings in a virus. Database must be kept up to date.
    - As viruses can camouflage themselves, many heuristics are employed, too
  - These tools are not perfect (and once an infection has occurred, it may be too late anyway)
  - Hence, basic "hygiene" as described above also goes a long way towards securing a system

Trojans

- Named after the Trojan horse in Homer's Ilias
  - Disguise as a useful (often tempting) software
  - Old: A simple calculator program destroyed all the data for a CBS executive in 1990
  - They are executed with good intentions and can install/download further software
  - May try to escalate their privileges by exploiting vulnerabilities on the target system

Drive-by downloads

- Name of a class of exploits that make use of browser vulnerabilities
- Early browsers, in particular, were famous for such vulnerabilities
- A simple visit to a website can lead to infection of the browser (i.e. take-over of the control flow, or execution of a program)
- That could then lead to further installation of unwanted programs and take-over of the machine

- The defence is to keep the browser up-to-date
  - Today's browsers **routinely** use auto-update mechanisms
  - Enable them – it has reduced the number of cases massively

Worms

- Malware with a specific way of spreading: replication via a network (as opposed to taking over a program)
- The first was released by Robert Morris in 1988.
  - convicted and given community service sentence
  - now a professor at MIT

- The worm was a program that was meant to travel slowly around the network gathering size information.
- The worm itself had an unforeseen effect which meant that it propagated as fast as possible and overloaded the infected machines.
- Large proportion of the internet of the time was infected and crashed.

War story: Morris Worm

- It exploited three well known flaws in Unix at that time to locate machines it could propagate to.
  - Guided search on Unix password file
  - Buffer overflow exploit on *fingerd* program
  - Trapdoor in the *sendmail* program
- Since then worms have become more virulent (e.g. Code Red) and are a very common infection vector for botnets.

Morris worm propagation

- A key technique used by the worm to propagate was a buffer overflow attack against the “fingerd” daemon on VAX systems.
- A special, long string was sent to the ‘finger’ service that caused fingerd to execute the worm code and create another copy of the worm.
- The worm then used the ‘hosts’ file to discover other machines on the net and try and infect them.
- Once the worm started to spread it was very difficult to stop.

Keyloggers

- Class of malicious code – malware
- A keylogger intercepts every key press and can save the information and send it to the attacker.
- A keylogger can examine the stream of keystrokes and capture passwords typed into web forms for banking sites, for example.
- Interestingly, keyloggers exist as hardware implementations, too!
  - Tiny USB keys that you can plug in without the victim noticing
  - Can even clip into the keyboard or monitor cable

Backdoors

- Sometimes the author of a software module may leave some extra "features" in the code that are not documented
  - typically for debugging purposes
  - provides a backdoor entry that bypasses normal controls
  - "Windows Golden keys" – August 2017

- Attackers may be able to take advantage of these features to gain control
  - Show debug stats
  - Dump to log file
  - Load configuration file

Rootkits

- Rootkits are, in principle, backdoors that go to great lengths to
  - Evade discovery
  - Execute with root privileges
- Some are **well intentioned:**
  - Sony XCP, to prevent illegal file copies
  - But soon became exploited as a vulnerability
- **Accepted wisdom** over decades of computer research is that every well-intended backdoor eventually gets exploited.
  - Hence, be wary of promises that some backdoor will never get found, exploited, etc. and is secure

Logic bombs

- Logic bombs are a superclass of exploits
  - Code that hides in normal software and becomes active when certain conditions are met, unknown to the intended user
    - E.g. visit to some website, trigger on a certain date etc.
  - Trojans can contain logic bombs
  - Insider attacks are feared:
    - E.g. disgruntled employee leaves logic bomb that activates one year after they were fired
- Very hard to find!
  - Automated analysis of programs for their logical, semantic meaning is impossible in the general case

Bot networks

- Bots are programs that participate in malicious activity
- They are often implemented as trojans or worms, making them spread via the network
- Infected computers communicate with a Command & Control centre and receive instructions
  - This creates a network of bots awaiting instructions
  - Often used for Distributed Denial-of-Service attacks: massive number of bots are asked to connect to a remote machine (e.g. website). Load brings it down.
- Serious business: organized crime rents out botnets to "customers"
- Botnets are commonly targeted by law enforcement:
  - Go after operators if you can identify them
  - Try to go after command & control centre, but can be hidden well

War story: Stuxnet

- Targeted SCADA systems (infrastructure) in June 2010.
  - Caused substantial damage to Iran's nuclear program.
- Exploited four zero day vulnerabilities in Microsoft Windows.
  - SMB vulnerability (MS08-067)
  - Print spooler vulnerability (MS10-061)
  - LNK vulnerability (CVE-2010-2568)
  - WinCC hardcoded password
- Spread via many routes, including USB keys
- Believed to be a state-sponsored attack

War story: Stuxnet

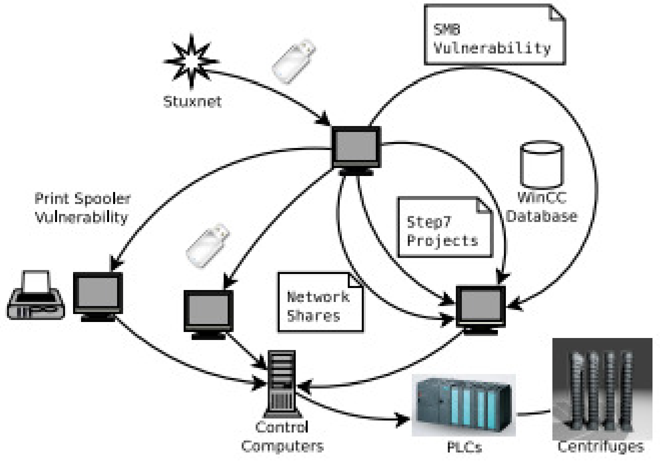
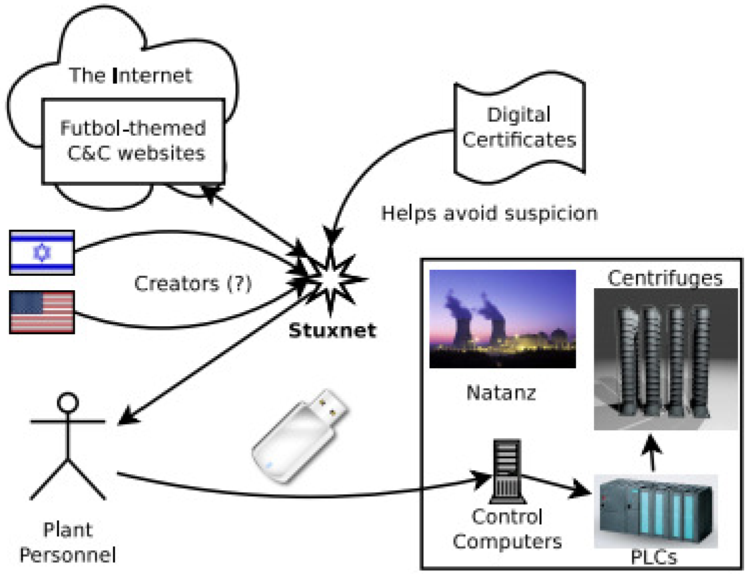


Figure 4: Stuxnet employs many ways to reach its target PLCs.



General defence principles

- Users:
  - Never work at higher privilege than necessary
  - Least privilege: do not work as admin
- Developers:
  - Check inputs rigorously
  - Test
  - Securely clear memory (password, sensitive info etc.)
  - Failsafe defaults
  - Simplicity: good design
- Operators:
  - Avoid *monoculture* if possible
  - same product/software
  - Configuration management (e.g system for updates)