BUFFER OVERFLOWS

Martin Read

Lecture aim

Introduction to buffer overflow

Lecture Objectives

1. What happens when you don't follow secure software development?

2. Buffer overflows, heap overflows …

Both sides, attacker & defence, need to know what other side is doing if they want to be effective…

- Practical next week

SECURE BY DESIGN?

- To avoid software vulnerabilities, a need to adopt secure software development practices throughout the software lifecycle

SOME EXAMPLES

- Fencepost errors (off-by-one errors)
  - Example: how many numbers do you need to process between the range M and N, when M=5 and N=17?
  - Example: OpenSSH channel allocation could result in a user gaining full privileges


```
if (id < 0 || id > channels_alloc) (1)
if (id < 0 || id >= channels_alloc) (2)
```

- Rapid functionality expansion often leads to vulnerabilities
  - Example: IIS Webserver support for Unicode
- Memory corruption

Software Vulnerability

1. Buffer overflows
   - software attempts to write data past end of size given
2. Bad input sanitation
   - software doesn't check if input valid
3. Race condition
   - software executes something "quickly" to change execution order
4. Access control
   - who is allowed to access/alter what
5. Weak authentication/authorization/cryptography
6. Control flow
   - altering data/pointers to change flow of the software

Software Vulnerability - categories

1. Memory corruption
   - accessing memory in way developers didn't intend
2. Injection
   - addition of unexpected data, pointers etc
3. Broken authentication
   - bad control access, authentication, encryption, etc.

Attack surface describes where the vulnerabilities that can be exploited are…

Normally means the external factors outsiders have access too (public websites, place to input information). If surface bypassed, then attacker can exploit internal vulnerabilities as well

Memory Corruption example

A file stores everyone's first name & birth year :

[….Alice1995……Bob1996]

There are 9 characters for first name

dots are used to fill gap at the front

Year is 4 characters

Theoretically, you are only allowed to alter your own data

While size of year is checked, does not check the size of your first name

1. What/how should Alice change Bob's name to "NotAlice"?
2. What/how should Alice change Bob's birth year to "1896"?

Memory Corruption example

A file stores everyone's first name & birth year :

[….Alice1995……Bob1996]

1. What/how should Alice change Bob's name to "NotAlice"?
   Change "Alice" to "….Alice1995.NotAlice"
   Need dots or else would shift

2. What/how should Alice change Bob's birth year to "1896"?
   Change "Alice" to "….Alice1995 ……Bob18" Need dots or else would shift

Memory Corruption Vulnerabilities

Memory corruption involves tricking a program to run arbitrary code that has been smuggled into memory

Affects stacks & heaps primarily, in some cases can effect other parts of memory

- Buffer Overflows
- Format Strings

Buffer Overflows

- High level languages assume programmer responsible for data integrity
  - no inbuilt functionality to check that contents of a variable can fit into the allocated memory space
  - condition can cause buffer overflow vulnerabilities

- Why not design compilers to be responsible for data integrity?

Report on Buffer Overflows in the MS Windows Environment

- https://www.ma.rhul.ac.uk/static/techrep/2009/RHUL-MA-2009-06.pdf

Buffer Overflow: A Well-Known Problem

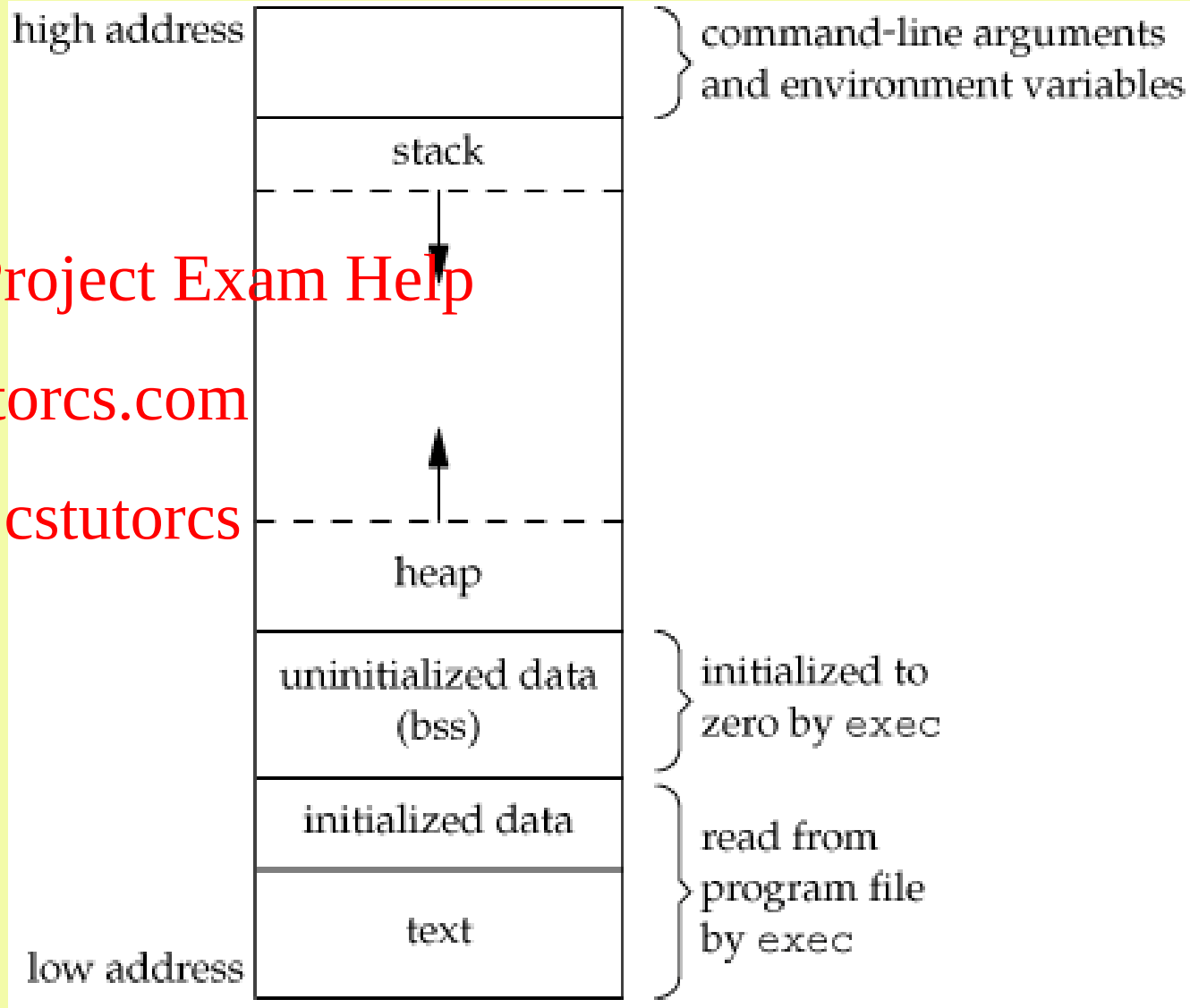
- A very common attack mechanism
  - 1988 Morris Worm, Code Red, Slammer, Sasser & many others
- Prevention techniques are known
- Still of major concern due to:
  - legacy of widely deployed buggy code
  - continued careless programming techniques

Buffer Overflow Basics

- Caused by programming error
- Allows more data to be stored than capacity available in a fixed sized buffer
  - buffer can be on stack, heap, global data
- Overwriting adjacent memory locations
  - corruption of program data
  - unexpected transfer of control
  - memory access violation
  - execution of code chosen by attacker

Reminder

1. **Stack**: function parameters, return addresses & local variables of function stored
2. **Heap**: All dynamically allocated memory here
3. **%eip** Instruction pointer register stores next instruction address
4. **%esp** Stack pointer regstr stores stack top address
5. **%ebp** Base pointer regstr keeps track of function variables

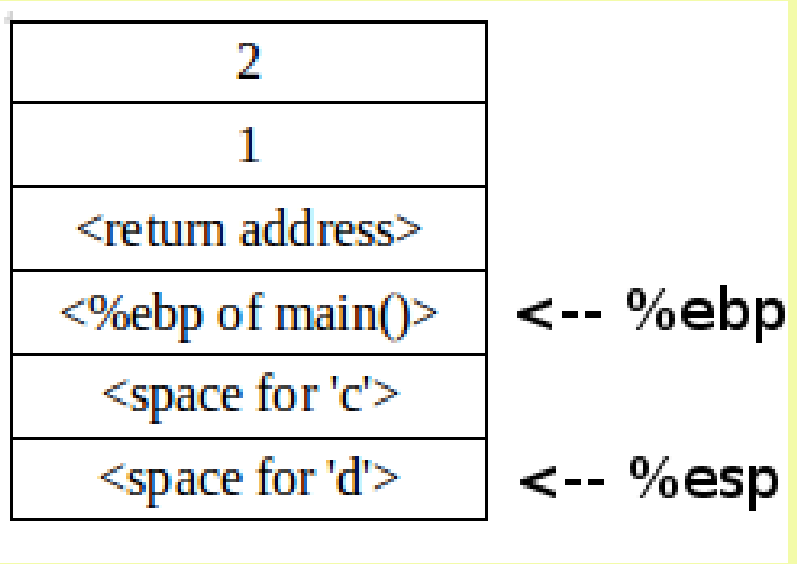


Buffer Overflows

- A buffer overflow/buffer overrun
  - anomaly where a program, while writing data to a buffer, overruns buffer's boundary & overwrites adjacent memory locations
- Buffers are created to hold a defined amount of data
  - overflow occurs when a program attempts to write more data to a fixed length block of memory (buffer) than it is allocated to hold
- We could overwrite data

 - but data in stack is not always strings & integers

- One popular attack is to rewrite function return addresses to change control flow



Example

- There is no possible control path to secretFunction()
- If we can rewrite the return address of echo() to secretFunction instead of main(), can alter flow
- User has control of input
- No buffer length checks

Should be on lab VM


```c
#include <stdio.h>

void secretFunction()
{
    printf("Congratulations!\n");
    printf("You have entered in the secret function!\n");
}

void echo()
{
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main()
{
    echo();

    return 0;
}
```

Buffer Overflow Attacks

To exploit a buffer overflow an attacker:

- Must identify a buffer overflow vulnerability
  - inspection, tracing execution, fuzzing tools
- Understand how buffer is stored in memory & determine potential for corruption

- Discovering vulnerabilities can be relatively easy
- Exploiting them to a desired effect requires experimentation
  - Experimenting with BASH & Perl at command line can be useful to generate overflow buffers on the fly

$ perl –e ‘print “A” x 20;’

–e: executes command

print: prints character A 20 times

$ perl –e ‘print “\x41” x 20;’

print: prints character A (ascii 0x41) 20x

$ perl –e ‘print “A”x20 . “BCD” . “\x61\x66 \x67 \x69”X2 . “Z”;’

. : concatenates strings/characters print: prints AAAAAAAAAAAAAAAAABCDafgiafgiZ’

$ $(perl –e ‘print “uname”;’)

To execute a shell command like a function, returning an output, surround command with () & prefix with $

Output of perl –e ‘print “uname”;’ will be executed

Heap overflow

- Each process has a heap & stack for execution
  - Volatile, dynamically allocated memory for program needs
  - Grows towards higher memory addresses

- Heap overflow/heap overrun is a type of buffer overflow

- Exploitation performed by corrupting data in ways to cause the application to overwrite internal structures, such as linked list pointers

- Heaps are complicated, changing in size, things get added, deleted & shifted
- We won't go into it, but since heaps are complex, with lots of pointers, there are lots of vulnerabilities

Language vulnerabilities

- Modern high-level languages have strong notion of type & valid operations
  - not vulnerable to buffer overflows
  - incurs overhead & some limits on use
- C & related languages have high-level control structures
  - **but** allow direct access to memory
  - hence vulnerable to buffer overflow
  - a large legacy of widely used, unsafe & hence vulnerable code

Insecure C functions

- Most vulnerabilities in C are related to buffer overflows & string manipulation
- In most cases, this would result in a segmentation fault, but specially crafted malicious input values, adapted to the architecture & environment could yield to arbitrary code execution

strcpy does not check buffer lengths & may overwrite memory zone contiguous to the intended destination

The whole family of functions is similarly vulnerable:

strcpy, strcat & strcmp

Secure C functions - mitigation

- Use **strlcpy,** if available (only the case on BSD systems)
  - However, it is very simple to define yourself…
- OR **strcpy_s**() - similar to strcpy(), when there are no constraint violations
  - function copies characters from source string to a destination character array up to & including terminating null character
- **gets()** does not check for buffer length
  - use fgets (& dynamically allocated memory)
- Other C function vulnerabilities
- **String formatting attacks**: printf, fprintf, sprintf & snprintf
  - next weeks lecture

Non Executable Address Space

- Many Buffer Overflow attacks copy machine code into buffer & transfer control to this
- Use virtual memory support to make some regions of memory non-executable (to avoid execution of attacker's code)
  - e.g. stack, heap, global data
  - need h/w support in MMU - long existed on SPARC/Solaris systems
  - more recently on x86 Linux/Unix/Windows systems
- Mapping from virtual to physical addresses handled by MMU chip in conjunction with OS
  - Provides translation of addresses for programs & a large memory space, but also provides protection & reduces memory fragmentation

Memory corruption prevention

Secure software practices to prevent this include:

1. Check size of object you are writing to & size of what you are writing

2. If you are taking information from one data type to another, check sizes

3. This is more difficult at the Assembly level (pro's/con's working close to hardware)

Preventing Memory Corruptions

1. Develop code to check sizes before anything is written
2. Lock data
   - Effective at one level but makes life harder in most cases
3. Make the "gap filler" less predictable? (Canaries)

These cover several stages in the software lifecycle

1. Code development for applications etc.
2. Policies put into place during code development/execution
3. Code development & computer organization stage of OS/Kernel/etc.

More Countermeasures

Canary-based approach

1. Place random number in memory
2. Check random number before performing action
3. If random number changed an overflow has occurred

Obfuscation of memory addresses (e.g.: PointGuard encryption)

Address Space Layout Randomization (compiler's job)

1. Randomizes base addresses of stack, heap, code & shared memory segments
2. Makes it harder for an attacker to know where in memory his code is located

Instruction Set Randomization

Compile-Time Defences: Programming Language

- Use a modern high-level languages with strong typing
  - not vulnerable to buffer overflow
  - compiler enforces range checks & permissible operations on variables
- Does have cost in resource use
- And restrictions on access to hardware
  - so still need some code in C-like languages

Compile-Time Defences: Safe Coding Techniques

- If using potentially unsafe languages e.g. C
- Programmer must explicitly write safe code
  - by design with new code
  - ***extensive after code review*** of existing code, (e.g., OpenBSD)

- Buffer overflow safety a subset of general safe coding techniques
- Allow for graceful failure (know how things may go wrong)
  - check for sufficient space in any buffer

Compile-Time Defences: Language Extension, Safe Libraries

- Proposals for safety extensions (library replacements) to C
  - performance penalties
  - must compile programs with special compiler
- Several safer standard library variants
  - new functions, e.g. strlcpy()
  - safer re-implementation of standard functions as a dynamic library, e.g. Libsafe

Summary

- Introduced basic buffer overflow attacks
- Stack/Heap buffer overflows
- Defences
  - compile-time, run-time
- Shellcode (not covered)
- Other related forms of attack (not covered)
  - replacement stack frame, return to system call, global data overflow