Coding: Buffer Overflows

David Basin

Department of Computer Science
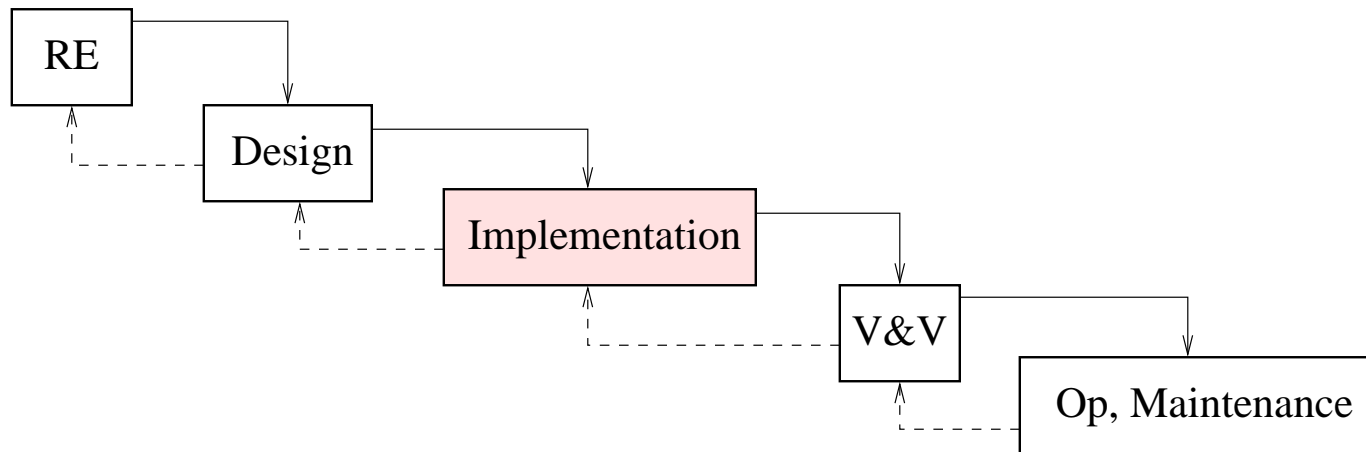ETH Zurich

Road map[1]

☞ **Motivation**

- Pointers in C

- Compilation and memory layout

- Buffer overflows

- Defense

- Summary

---

[1]Some of the material from this talk was developed by Heiko Mantel, for Security Engineering 2004/05

Context: implementation

It's possible to have a great design but an insecure implementation

1. Implementation deviates from design

2. Design leaves many options open (including unsafe ones)

   E.g., design doesn't specify password lengths, their storage, ...

3. Concretization may introduce new vulnerabilities

   E.g., buffer overflows and other attacks on data and control

Motivation — buffer overflows

- Public enemy #1

  See e.g., CERT advisories or SANS top 20 vulnerabilities ranking

- Example: Morris Internet Worm (November 1988)

  ▶ Attacked Sun3 and VAX systems running BSD

  ▶ Fairly clever, e.g., named itself "sh" (hard to detect) and set maximum core-dump size to 0 bytes (hard to catch)

  ▶ Propagated itself using various exploits, including a buffer overflow attack against the finger daemon *fingerd*.

  ▶ Did not delete files, but resulting financial loss between $100,000 and $1,000,000 (US General Accounting Office)

Role of studying vulnerabilities (and a warning)

- Understanding vulnerabilities is vital to developing and evaluating countermeasures. Failure to consider them leads to insecure code or overconfidence in partial solutions

- Exploiting vulnerabilities (= hacking) is a criminal offense

- We expect you to use this knowledge in a responsible and ethical way

- Any experimentation must be done in a controlled environment

Buffer overflows

- A buffer is a contiguous region of memory storing data of the same type, e.g., characters

- A buffer overflow occurs when data is written past a buffer's end

- The resulting damage depends on

  ▶ Where the data spills over to
  ▶ How this memory region is used (e.g., flags for access control)
  ▶ What modifications are made

- Example: Morris attack on finger

  ▶ Intended use: finger basin@inf.ethz.ch
  ▶ Abuse: finger ⟨exploit-code⟩ ... ⟨return-address⟩
  ▶ Overwrites the return address and provides exploit code

Road map

- Motivation

☞ **Pointers in C**

- Compilation and memory layout

- Buffer overflows

- Defense

- Summary

C primer

- C (and C++) is powerful, widespread, and undisciplined!

- Basic data types

  Char (1 byte), int ($\geq$ 2 bytes), long ($\geq$ 4 bytes) ...

- Pointers: expressions like int *ptr

  - ▶ ptr is the address of a memory cell
  - ▶ *ptr is the contents of the memory cell

- Address taken using &. For i and integer and f a function,

  - ▶ &i is the address of the cell storing i
  - ▶ &f is the address of a function f

Pointers — examples

```
int i = 0;
int *ptr = NULL;


ptr = &i;


*ptr = 15;


ptr = &ptr;


ptr++;
```
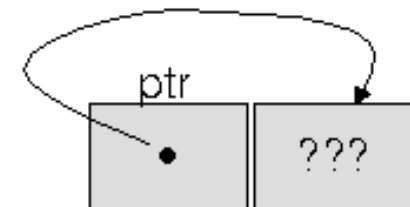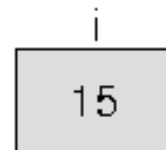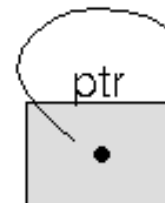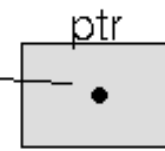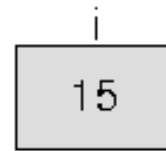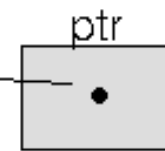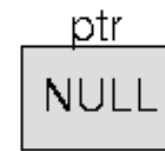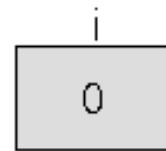
Arrays

- Declaring and accessing an array

  - ▶ `char buf[8];` declares a buffer for up to 8 characters
  - ▶ Elements are `buf[0]`, `buf[1]`, .., `buf[7]`
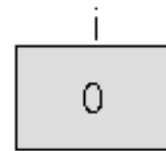 - ▶ No bounds checking, so `buf[13]` is also possible

- Arrays are similar to pointers: store address where buffer begins

- Some equalities

$$\&(buf[0]) = buf \qquad buf[0] = *(buf+0)$$
$$\&(buf[1]) = buf + 1 \qquad buf[1] = *(buf+1)$$

- Strings are arrays of characters, with `'\0'` at end

  - ▶ E.g., string `"hi"` represented by `'h'` `'i'` `'\0'`
  - ▶ No explicit information about length of strings
  - ▶ This savings is part of the problem!

Input, output and string handling functions

- `getchar()` reads a character from standard input

- `putchar(c)` writes a character to standard output

- `printf("Page %i has title %s\n", pageno, title)`

  ▶ String printed to standard output may contain place holders replaced by arguments
  ▶ Place holders provide formatting instructions.
    E.g., `%i` stands for an integer in decimal and `%s` for a string

- Various other functions store strings in buffers. E.g.

  ▶ `gets(dst)`   read string from `stdin` into `dst`
  ▶ `strcpy(dst,src)`   copy string `src` into buffer `dst`
  ▶ `sprintf(dst, <FmtStr>,<Exp>)`   print string into buffer
  ▶ `scanf, fscanf, ...`

A vulnerable program

buffer of
limited size

```
#include <stdio.h>
int main()
  {   char buf[8]                    // buffer for storing the input string
      char ch;                       // auxiliary variable
      char *ptr = buf;               // auxiliary pointer
      while ((ch=getchar()) != '\n'  // terminate on EOL
              && ch != -1)           // terminate on error
        {  *ptr = ch;                // store character
           ptr++;                    // increment pointer
        }
      *ptr = '\0';                   // terminate the string
      printf("%s\n",buf);
      return 0;
  }
```

is written
without limits

Some example runs

```
$ ./my-getstring
example
example
```

```
$ ./my-getstring
long example
long example
Illegal instruction
```

```
$ ./my-getstring
very long example
very long example
Segmentation fault
```
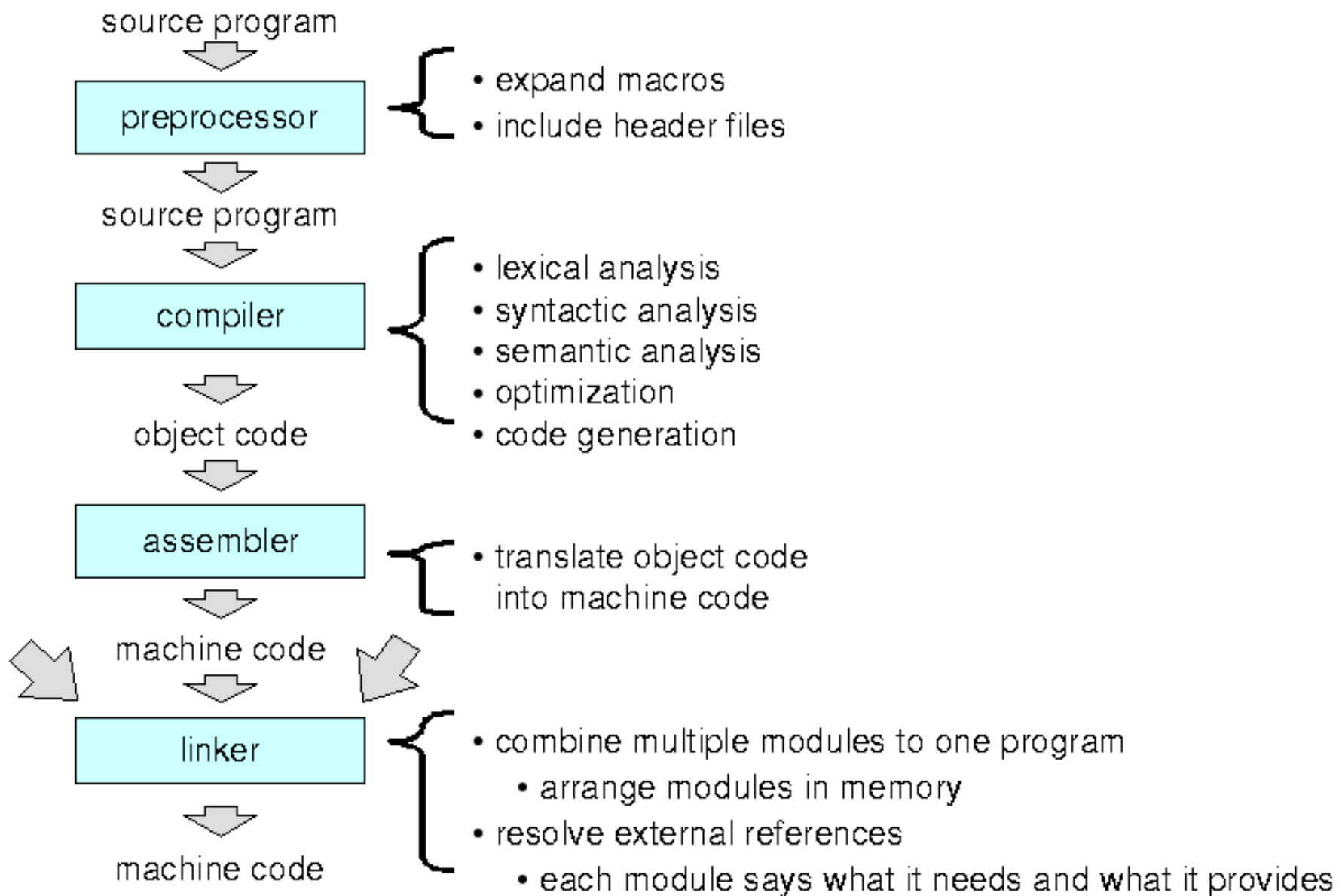
Road map

- Motivation

- Pointers in C

☞ **Compilation and memory layout**

- Buffer overflows

- Defense

- Summary

Compilation

source program

⬇

preprocessor	{	• expand macros
		• include header files

⬇

source program

⬇

compiler	{	• lexical analysis
		• syntactic analysis
		• semantic analysis
		• optimization
		• code generation

⬇

object code

⬇

assembler	{	• translate object code
		into machine code

⬇

machine code

⬇

linker	{	• combine multiple modules to one program
		• arrange modules in memory
		• resolve external references
		• each module says what it needs and what it provides

⬇

machine code

Loading

machine code

loader

machine code
in memory

- loading of program into the address space of the process

- Adapting addresses or initializing a relocation register (hardware supported relocation)

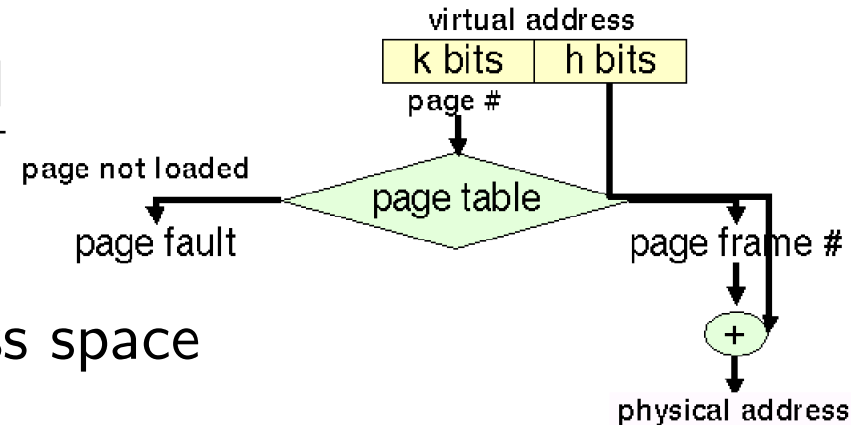- dynamic loading and linking of libraries for unresolved references

Recall how (virtual) memory is organized

- Memory named by virtual addresses

- Each process has its own virtual address space

- Protection: process can only access memory in its own space

virtual address

k bits | h bits

page #

page not loaded

page table

page fault

page frame #

+

physical address

A program in memory

```
$ gdb my-getstring
(gdb) disas main
Dump of assembler code for function main:

0x8048440   push   %ebp
0x8048441   mov    %esp,%ebp
0x8048443   sub    $0x18,%esp
0x8048446   lea    0xfffffff8(%ebp),%eax
0x8048449   mov    %eax,0xfffffff0(%ebp)
0x804844c   call   0x80482e8 <getchar>
0x8048451   mov    %eax,%eax
0x8048453   mov    %al,0xfffffff7(%ebp)
0x8048456   mov    0xfffffff7(%ebp),%al
0x8048459   cmp    $0xa,%al
0x804845b   je     0x8048478 <main+56>
0x804845d   cmpb   $0xff,0xfffffff7(%ebp)
0x8048461   jne    0x8048468 <main+40>
0x8048463   jmp    0x8048478 <main+56>
0x8048465   lea    0x0(%esi),%esi
0x8048468   mov    0xfffffff0(%ebp),%edx
0x804846b   mov    0xfffffff7(%ebp),%al
0x804846e   mov    %al,(%edx)
0x8048470   lea    0xfffffff0(%ebp),%eax
```

```
0x8048473   incl   (%eax)
0x8048475   jmp    0x804844c <main+12>
0x8048477   nop
0x8048478   mov    0xfffffff0(%ebp),%eax
0x804847b   movb   $0x0,(%eax)
0x804847e   sub    $0x8,%esp
0x8048481   lea    0xfffffff8(%ebp),%eax
0x8048484   push   %eax
0x8048485   push   $0x8048508
0x804848a   call   0x8048328 <printf>
0x804848f   add    $0x10,%esp
0x8048492   mov    $0x0,%eax
0x8048497   leave
0x8048498   ret
0x8048499   lea    0x0(%esi),%esi
0x804849c   nop
0x804849d   nop
0x804849e   nop
0x804849f   nop
```

Layout of virtual memory

Linux on Intel x86 family

- Stack grows downward
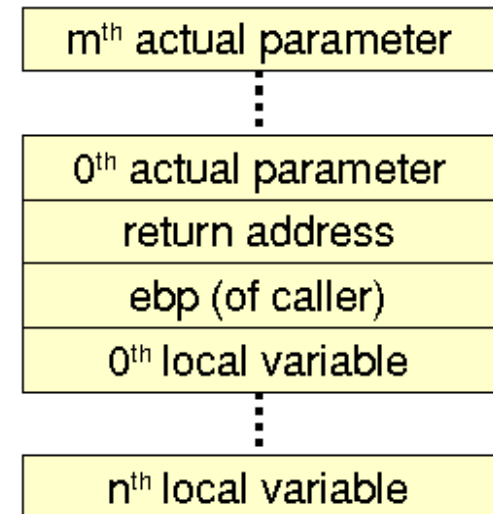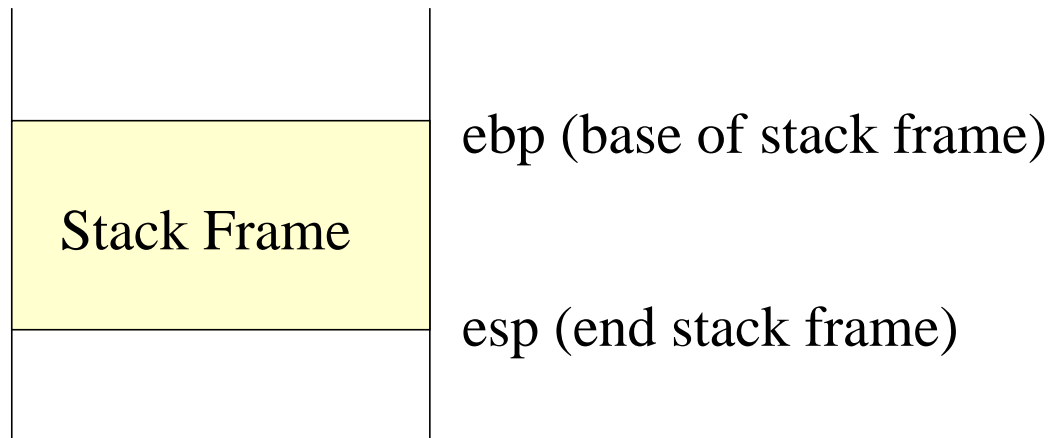
- Stack frame holds

  ▶ Calling parameters
  ▶ Local variables for functions
  ▶ Various addresses

- Heap grows upwards

  ▶ Dynamically allocated storage generated using alloc, malloc, ...

- Data: statically allocated storage

- Text: Executable code, read only

FFFFFFFF

stack

heap

data

text

00000000
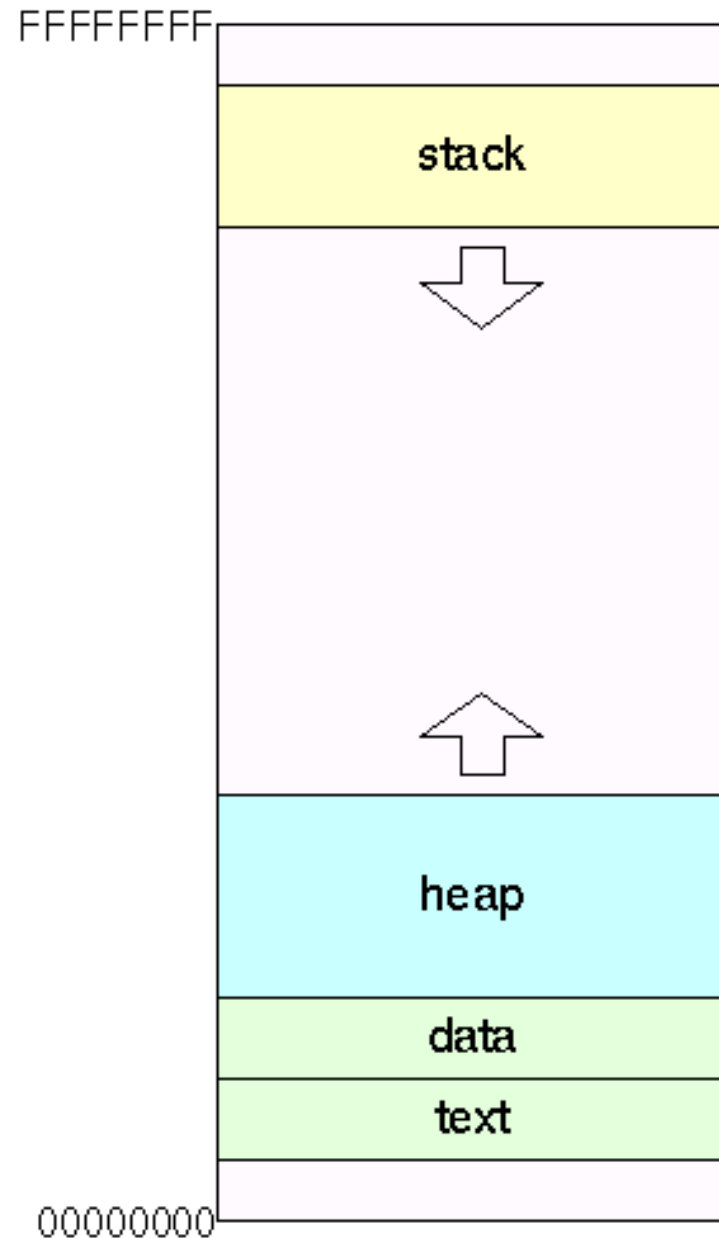
Structure of the stack

- Stack grows downwards: one stack frame per subroutine called

Higher values

	ebp (base of stack frame)
Stack Frame	
	esp (end stack frame)

m^{th} actual parameter
⋮
0th actual parameter
return address
ebp (of caller)
0th local variable
⋮
n^{th} local variable

Lower values

- Hardware registers contain:

  ebp (extended) base pointer register: start current frame
  esp (extended) stack pointer register: top (end) of stack

Another vulnerable program

```
int main (void) {
  char pw[8];
  sprintf (pw, "root-pw");
  if (authenticate(pw) > 0) {
    printf ("[root]$ ...\n");
  } else {
    printf ("Root: incorrect password\n");
  }
  return 0;
}
```

```
#include <stdio.h>
int authenticate (char* pw) {
  int result = 0;
  char buf[8];
  printf ("Root Password: ");

  if (gets(buf) !=0 ){
    if (strcmp (buf, pw) == 0) {
      result = 1;
    }
  }
  return result;
}
```

- $ ./my-authenticate

  Root Password: guess

  Root: incorrect password

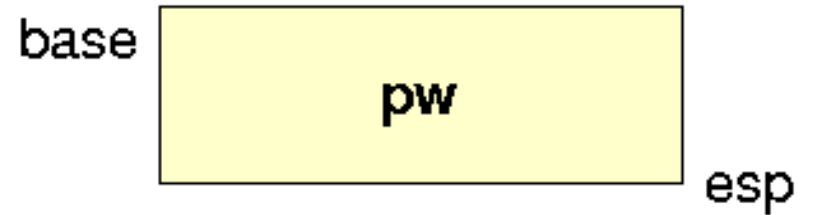- $ ./my-authenticate

  Root Password: root-pw

  [root]$ ...

Stack dynamics in example

1. start (of main)

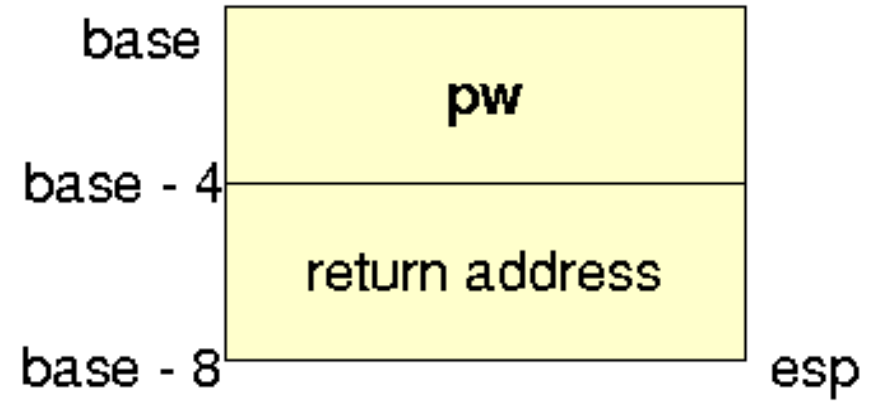base $\overline{\hspace{4cm}}$ esp

Stack dynamics in example

1. start (of main)
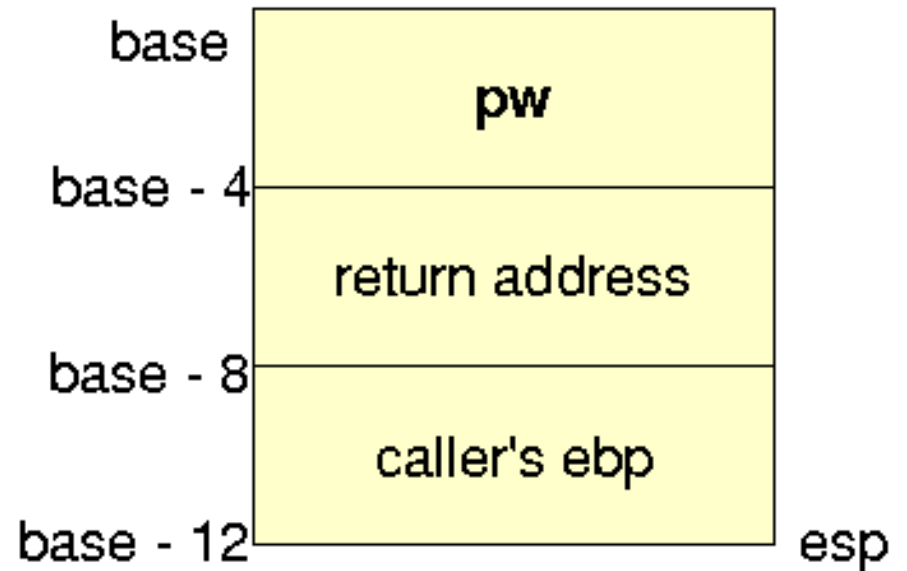
2. push argument pw onto stack

Stack dynamics in example

1. start (of main)

2. push argument pw onto stack

3. call authenticate (pushes return address onto stack)

Stack dynamics in example

1. start (of main)
2. push argument pw onto stack
3. call authenticate (pushes return address onto stack)
4. push caller's ebp onto stack

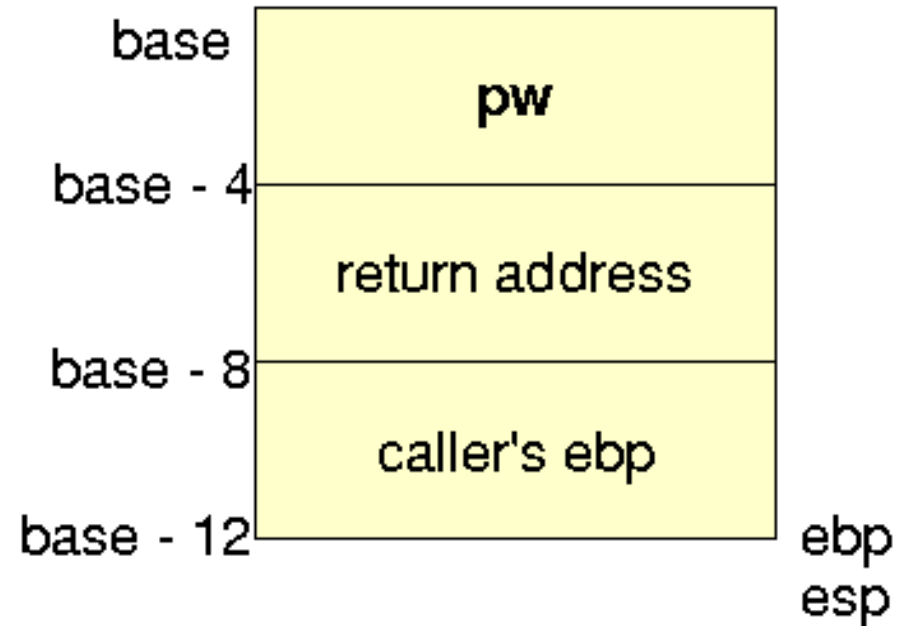# Stack dynamics in example

1. start (of main)
2. push argument pw onto stack
3. call authenticate (pushes return address onto stack)
4. push caller's ebp onto stack
5. copy esp into ebp

Stack dynamics in example

1. start (of main)
2. push argument pw onto stack
3. call authenticate (pushes return address onto stack)
4. push caller's ebp onto stack
5. copy esp into ebp
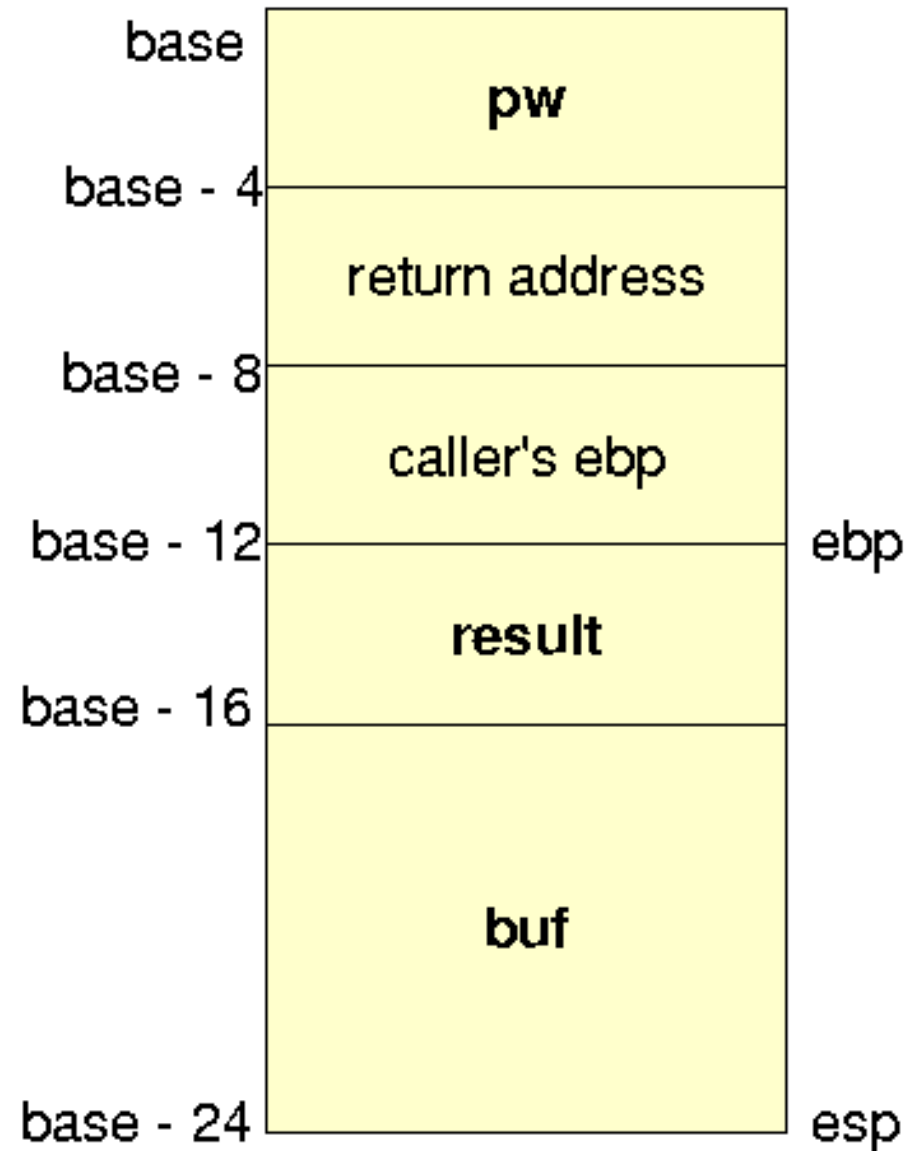6. decrement esp, creating space for local vars

Stack dynamics in example

1. start (of main)
2. push argument pw onto stack
3. call authenticate (pushes return address onto stack)
4. push caller's ebp onto stack
5. copy esp into ebp
6. decrement esp, creating space for local vars
7. Compute authenticate function (stack idle)

base

pw

base - 4

return address

base - 8

caller's ebp

base - 12 ebp
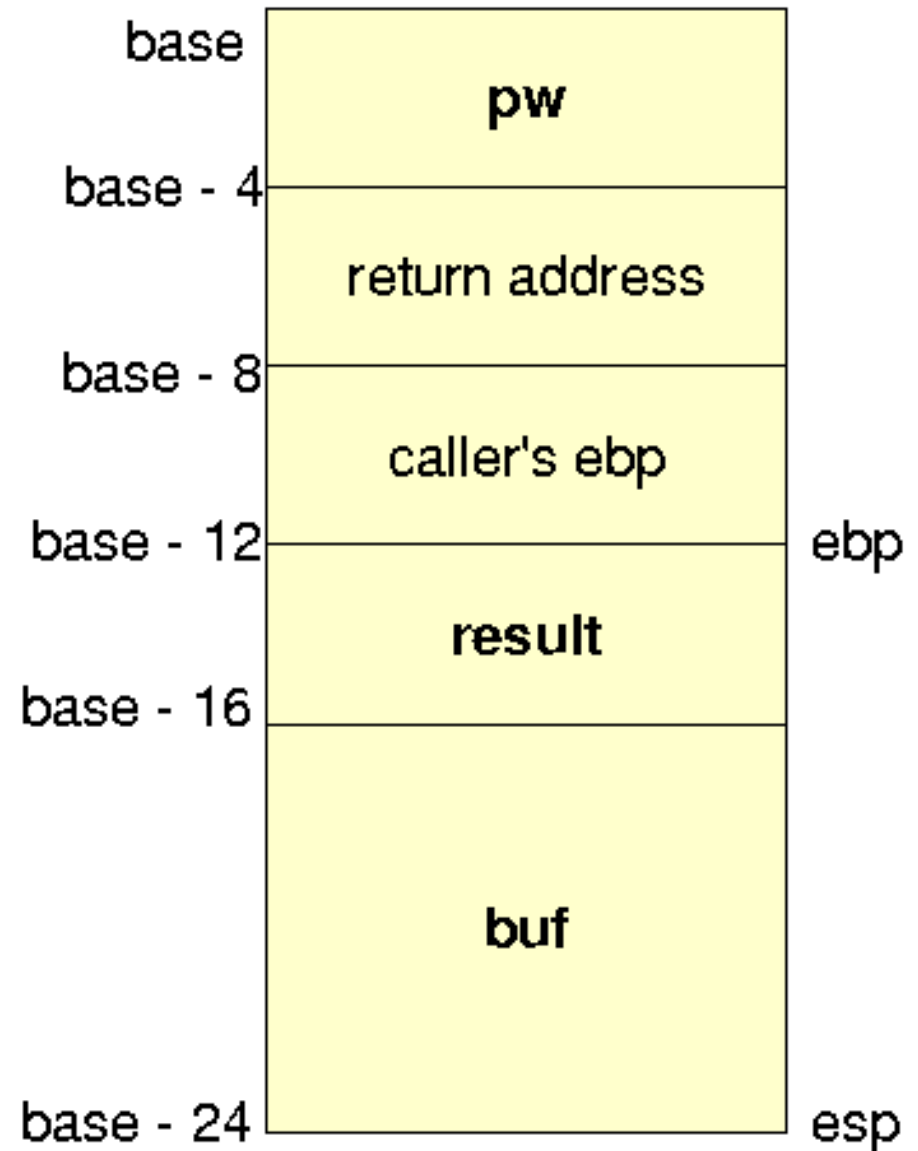
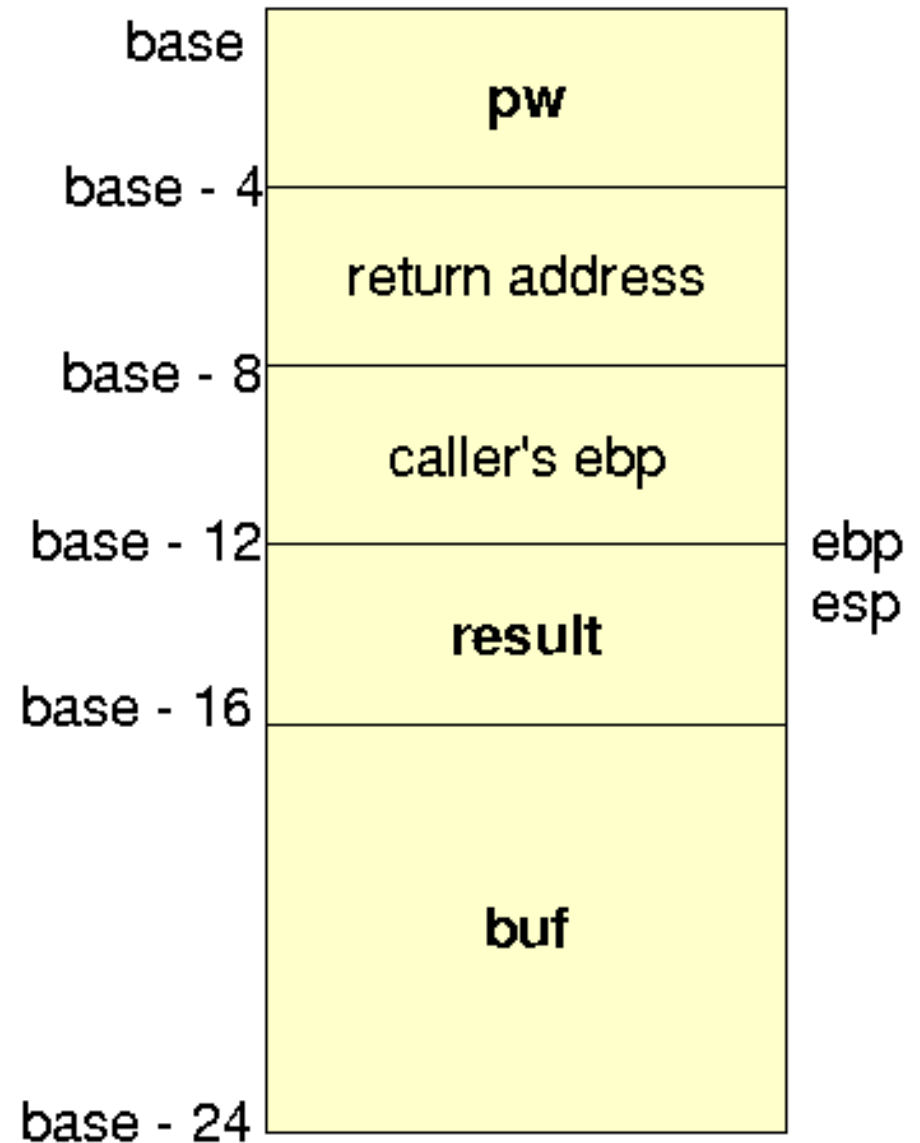result

base - 16

buf

base - 24 esp

Stack dynamics in example

1. start (of main)
2. push argument pw onto stack
3. call authenticate (pushes return address onto stack)
4. push caller's ebp onto stack
5. copy esp into ebp
6. decrement esp, creating space for local vars
7. Compute authenticate function (stack idle)
8. copy caller's ebp into esp

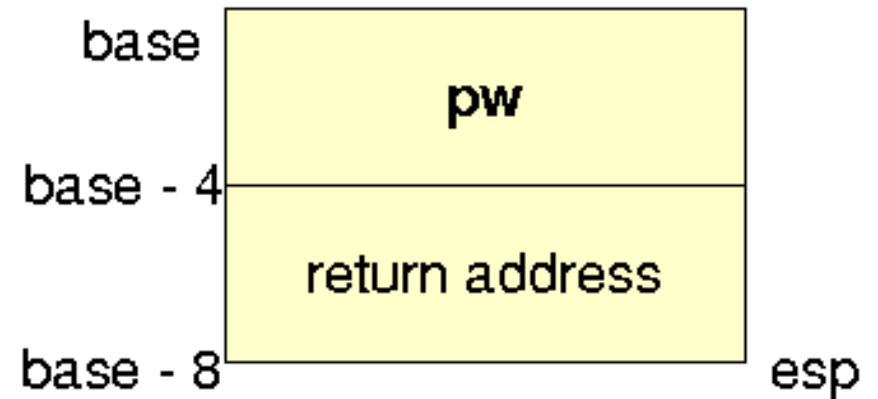|	
base	pw
base - 4	
	return address
base - 8	
	caller's ebp
base - 12	
	result
base - 16	
	buf
base - 24	

ebp
esp

Stack dynamics in example

1. start (of main)

2. push argument pw onto stack

3. call authenticate (pushes return address onto stack)

4. push caller's ebp onto stack

5. copy esp into ebp

6. decrement esp, creating space for local vars

7. Compute authenticate function (stack idle)

8. copy caller's ebp into esp

9. pop vars and caller's ebp from stack

base

pw

base - 4

return address

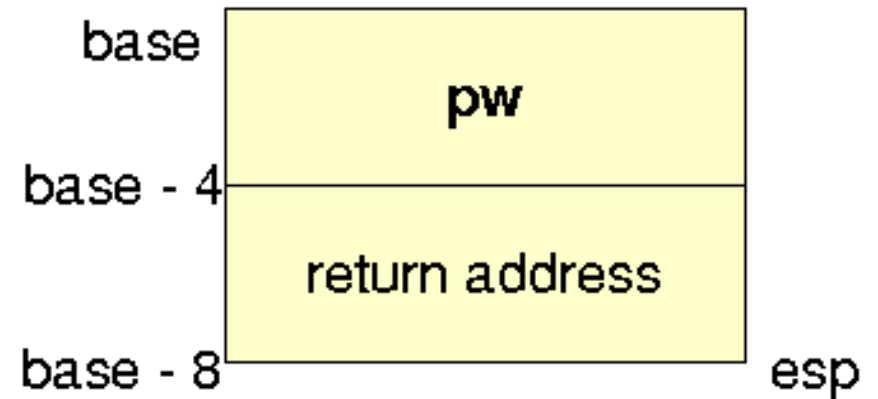base - 8   esp

Stack dynamics in example

1. start (of main)
2. push argument pw onto stack
3. call authenticate (pushes return address onto stack)
4. push caller's ebp onto stack
5. copy esp into ebp
6. decrement esp, creating space for local vars
7. Compute authenticate function (stack idle)
8. copy caller's ebp into esp
9. pop vars and caller's ebp from stack
10. return

Road map

- Motivation

- Pointers in C

- Compilation and memory layout

☞ **Buffer overflows**

- Defense

- Summary

Where is vulnerability in authenticate?

- Some non-problems

  ▶ Variables result and buf are initialized prior to use

  ▶ Programming logic ok. result is 1 iff input equals password supplied by pw

- The problem

  ▶ Implementation of gets does not prevent overflow of buf

  ▶ Stack layout

```c
#include <stdio.h>
int authenticate (char* pw) {
  int result = 0;
  char buf[8];
  printf ("Root Password: ");

  if (gets(buf) !=0 ){
    if (strcmp (buf, pw) == 0) {
      result = 1;
    }
  }
  return result;
}
```

Example: a long password

- Given a buffer overflow, data from buf spills over into result

- Can change value of result

  result initialized to 0 and updated only if authentication succeeds

- Works (conceptually) as follows
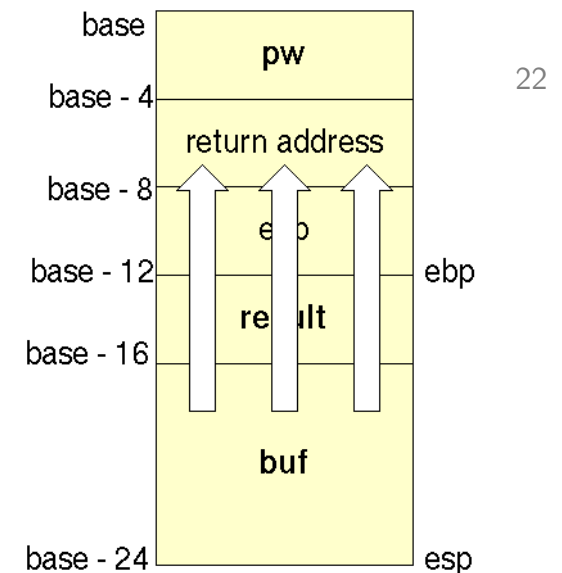
  ```
  $ ./my-authenticate
  Root Password: aaaaaaaa\x01\x00
  [root]$ ...
  ```

  buf filled with aaaaaaaa and hence x01 overwrites result.

- Violation of data integrity!

  ▶ Variables modified without explicit assignment
 ▶ Thinking purely on the level of C misses this possibility!

Long password (cont.)

- In reality, buffer overflows are a bit trickier

- There could be a gap between buf and result.

- If input is too long then ebp or return
  may be effected

- Entering hexadecimal values `\x01` and `\x00`

  Here one can enter input strings using a program via a pipe

Can we program defensively against such errors?
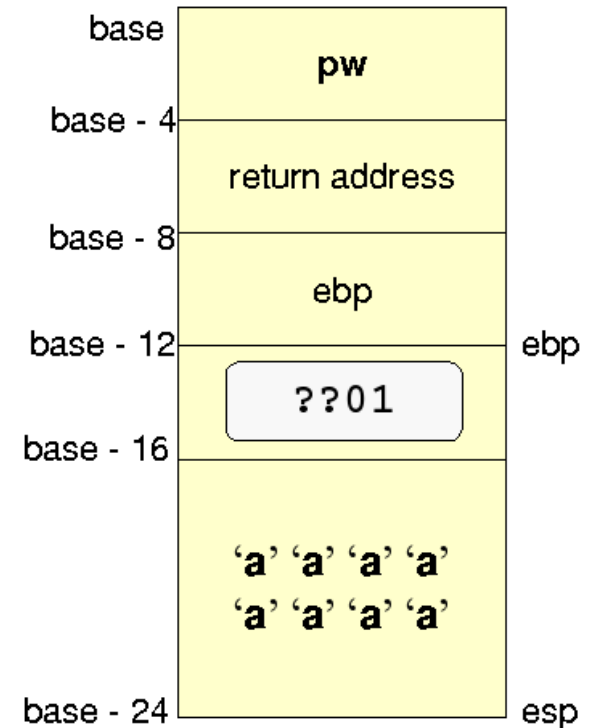
A defense: restoring variables

```
#include <stdio.h>
int authenticate (char* pw) {
  int result = 0;
  char buf[8];
  printf ("Root Password: ");

  if (gets(buf) !=0 ){
    if (strcmp (buf, pw) == 0) {
      result = 1;
    }
  }
  return result;
}
```

```
#include <stdio.h>
int authenticate (char* pw) {
  int result = 0;
  char buf[8];
  printf ("Root Password: ");

  if (gets(buf) !=0 ){
    if (strcmp (buf, pw) == 0) {
      result = 1;
    } else { result = 0; }
  }
  return result;
}
```

- Result is updated after gets

  Now result is correct even if buf overflows into result.

- But other dangers still remain!

  Would you recommend this defense?

Example: a very long password

- What happens on:

  ```
  $ ./my-authenticate
  Root Password:
  aaaaaaaaaaaa\xbb\xbb\xbb\xbb\x11\x47\x11\x47\00+
  ```
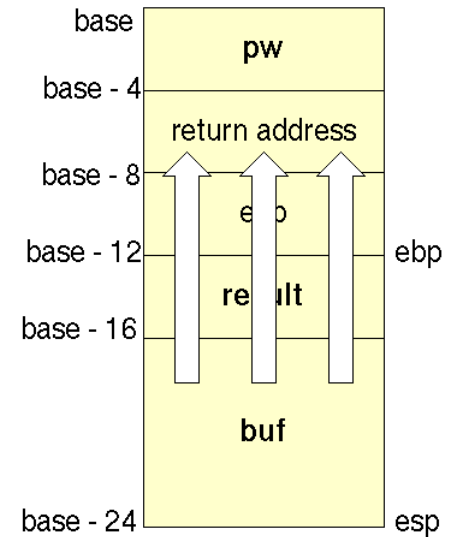

- Overwrites result, ebp, and return

  buf: "aaaaaaaa"
  result: "aaaa"
  stored ebp: bb bb bb bb
  return address: 47 11 47 11

- Call of ret (when leaving subroutine) pops
  47 11 47 11 off stack, starting execution at this memory location

Overflow has destroyed integrity of code-flow!

Exploit code (or where evil doers jump)

- Where would a malicious attacker jump to?
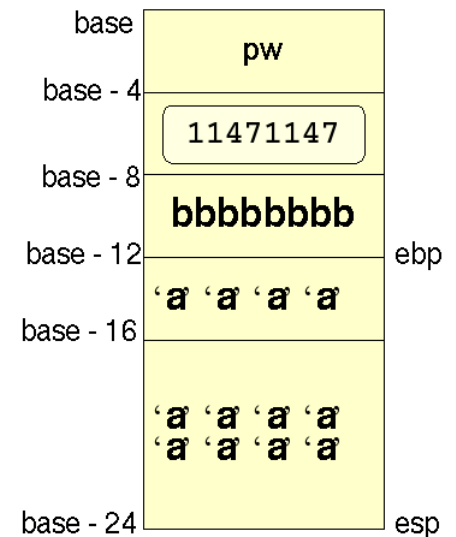
  One common target is to code that creates a (root-)shell

- Where in memory does this code go?

- Common approach: place exploit code on the stack

  Usually, place within the very buffer that is overflowed

- Return address must then point exactly to the exploit's entry point

  ▶ Non-trivial in practice.

  ▶ Trick used of starting exploit code with a "landing zone" of values representing No-op instructions

Exploit code (cont.)

- In above approach, exploit code and landing zone fit into buffer

- Alternatively, attacker places exploit code:

  - ▶ On the stack: into parameters or other local variables
  - ▶ On the heap: into some dynamically allocated memory region
  - ▶ Into environment variables (on stack)

- Another alternative is to abuse existing code

  E.g., jump to fragments of the program code or library functions

- Clever abuses have become a kind of hacker sport!

  For more on this, see Pincus/Baker article from 2004.

Road map

- Motivation

- Pointers in C

- Compilation and memory layout
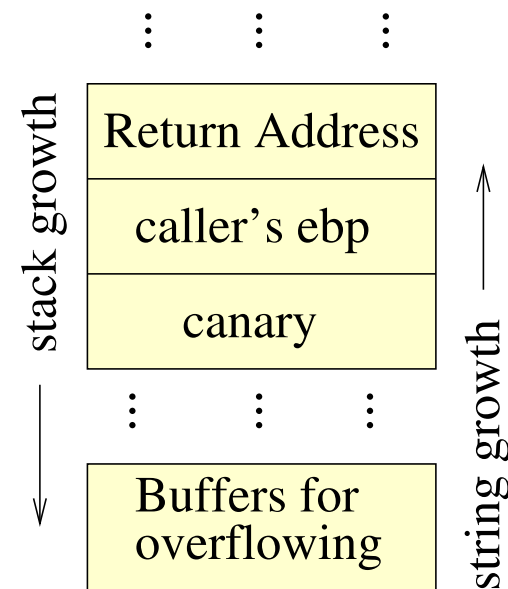
- Buffer overflows

☞ **Defense**

- Summary

Insert a canary

- A canary is a value on the stack whose value is tested before returning.

```
void function (...) {
    int canary;
    char buf[MAX_SIZE]; // other declarations follow

    canary = CANARY_VALUE;
    gets(buf);     // or some other dangerous operation
    ...
    if(canary != CANARY VALUE)
        exit
    return; }
```

- A canary is a random value (hard for attacker to guess) or a value composed of different string terminators (CR, LF, Null, -1).

- Implementations for GCC and Microsoft Visual C++ compilers

  Known attacks exist (see Pincus/Baker)

Stack diagram (right side):

stack growth ↓

| Return Address |
| caller's ebp |
| canary |

⋮ ⋮ ⋮

| Buffers for overflowing |

string growth ↑

Defensive programming

- Avoid unsafe library functions, e.g.,strcpy, gets, ...

  ▶ Replace with safe variants, e.g., str**n**cpy, **f**gets, ...
  ▶ E.g., replace `strcpy(dst,src)` with `strncpy(dst,src,dst_size-1)`.

- Always check bound of arrays when iterating over them

- Mechanisms for doing this

  ▶ Careful programming
  ▶ Audit teams
  ▶ Use grep or more sophisticated tools

- Limitations: error prone and audits are time consuming

Defensive programming (cont.)

BUGS

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.

— From gets(3) manual page

Automatic array bounds checking

- Approach: compiler automatically adds an explicit check to each access to an array

- Checks inserted during code generation

  Example: GCC enhancements of Jones&Kelly.

- Drawbacks

  ▶ It can be difficult to determine the bounds of an array
  ▶ Loss of performance can be substantial, e.g., factor 10-30!
  ▶ Some compilers only check explicit array references, like `buf[n]`, but not pointer references, like `*(buf+n)`

Avoid C (++)

- Use a language that is type safe

  ▶ Length of an array is part of its type
  ▶ Assigning contents of a buffer to a smaller buffer is a type error
  ▶ **Examples:** Pascal, Java, or ML

- Problems and limitations

  ▶ Often the choice of programming language is not yours
  ▶ Bugs still possible if run-time environment is programmed in an unsafe language
    Example: several buffer overflows vulnerabilities in implementations of the JAVA virtual machine
  ▶ C(++) has its advantages, in particular for writing efficient programs "close to the machine".

Avoid buffers on stack

- Avoid declaration of local array variables in functions.

- Instead, use heap storage, e.g., allocate space with `malloc()`.

- As return address is on the stack, it can not be overwritten by a buffer-overflow on the heap.

- Does this solve all our worries?

Avoid buffers on stack

- Avoid declaration of local array variables in functions.

- Instead, use heap storage, e.g., allocate space with `malloc()`.

- As return address is on the stack, it can not be overwritten by a buffer-overflow on the heap.

- Does this solve all our worries?   No!

  ▶ Heap overflows are also a real problem (not covered here)
  ▶ While they have no effect on control-flow integrity, they can violate data integrity.

Non-Executable Buffers

- Mark stack [or heap] as being non-executable.

  $\Rightarrow$ attacker cannot run exploit stored in buffers on stack [heap].

- Mechanisms

  ▶ Extend OS with a register storing maximal executable address
  ▶ Alternatively, tag pages as (non)executable in the page table

- Problems and limitations

  ▶ Attacker can still execute code in the text segment
  ▶ Attacker can still violate data integrity
  ▶ Sometimes too restrictive
    Unix signal handlers usually execute on the current process
    stack. This lets the signal handler return to the point that
    execution was interrupted in the process.

Road map

- Motivation

- Pointers in C

- Compilation and memory layout

- Buffer overflows

- Defense

☞ **Summary**

Conclusions and lessons learned

- Arrays in C can be over-written.

- The result is that data and control flow can be altered in ways not described by program itself.

- This is a massive problem and has been so for many years!

- Defense is paramount!

  ▶ Program defensively: only use or write functions that do bounds checking.
  ▶ Carefully weigh pros and cons of programming language used. Do advantages of C outweigh its disadvantages?
  ▶ Compiler/hardware support for preventing overflows is available. It helps, but be aware of the limitations and overhead.

Literature

- John Viega, Gary McGraw. Building Secure Software. Addison-Wesley, 2002.

- Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, Jonathan Walpole. *Buffer Overflows: Attacks and Defences for the Vulnerability of the Decade.* DARPA Information Survivability Conference and Exposition, 2000.

- Jonathan Pincus, Brandon Baker. *Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns*, IEEE Security & Privacy, 2004.

- AlephOne. Smashing the Stack for Fun and Profit. 1996