CSE 127: Buffer Overflow (Continuation)

George Obaido, Ph.D.

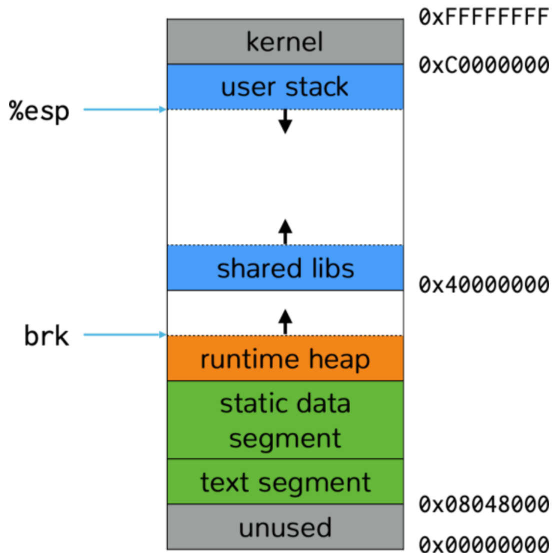
UCSD

Spring 2022 Lecture 2

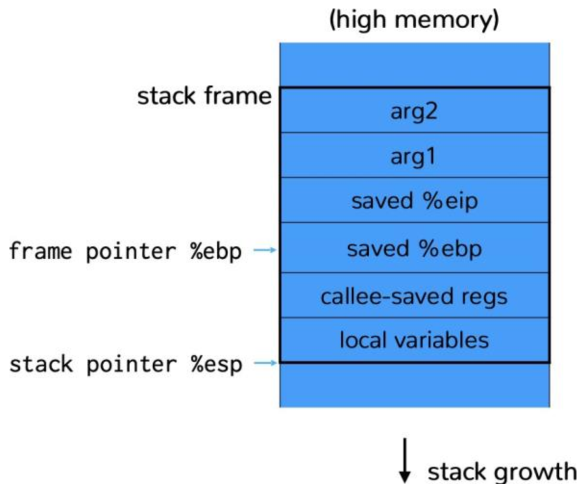
Continuation

Linux process memory layout

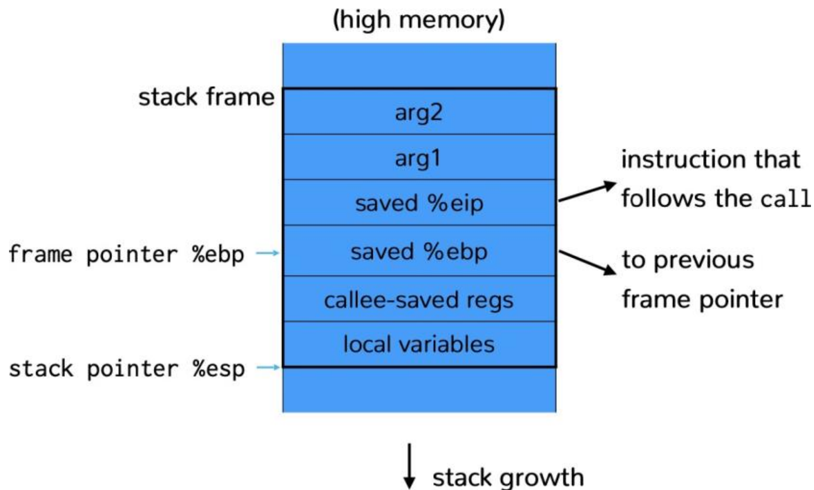
- **Stack:** Stores local variables.
- **Heap**: Dynamic memory for programmer to allocate.
- **Data segment**: Stores global variables, separated into initialized and uninitialized.
- **Text segment**: Stores the code being executed.



The stack



The stack



The Stack

- Stack divided into frames
  - Frame stores locals and args to called functions
- Stack pointer points to top of stack
  - x86: Stack grows down (from high to low addresses)
  - x86: Stored in `%esp` register (`%rsp` on 64-bit)
- Frame pointer points to caller's stack frame
  - Also called base pointer
  - x86: Stored in `%ebp` register (`%rbp` on 64-bit)

Brief review of x86 assembly

- We're going to use AT&T/gasm syntax
  - `op src, dst`
  - `%register     $literal     offset(memory-reference)`

Brief review of x86 assembly

- We're going to use AT&T /gasm syntax
  - `op src, dst`
  - `%register     $literal     offset(memory-reference)`

- Examples:

movl %eax, %edx →

Brief review of x86 assembly

- We're going to use AT&T /gasm syntax
  - `op src, dst`
  - `%register` `$literal` `offset(memory-reference)`

- Examples:

Assembly		**C Pseudo-code**
movl %eax, %edx	→	edx = eax

Brief review of x86 assembly

- We're going to use AT&T /gasm syntax
  - `op src, dst`
  - `%register     $literal     offset(memory-reference)`

- Examples:

Assembly		C Pseudo-code
movl %eax, %edx	→	edx = eax
movl $0x123, %edx	→	

Brief review of x86 assembly

- We're going to use AT&T /gasm syntax
  - `op src, dst`
  - `%register     $literal     offset(memory-reference)`

- Examples:

```
movl %eax, %edx       →   edx = eax
movl $0x123, %edx     →   edx = 0x123
```

Brief review of x86 assembly

- We're going to use AT&T /gasm syntax
  - `op src, dst`
  - `%register` `$literal` `offset(memory-reference)`

- Examples:

movl %eax, %edx → edx = eax

movl $0x123, %edx → edx = 0x123

movl (%ebx), %edx →

Brief review of x86 assembly

- We're going to use AT&T /gasm syntax
  - `op src, dst`
  - `%register` `$literal` `offset(memory-reference)`

- Examples:

```
movl %eax, %edx        →  edx = eax
movl $0x123, %edx      →  edx = 0x123
movl (%ebx), %edx      →  edx = *((int32_t*) ebx)
```

Brief review of x86 assembly

- We're going to use AT&T /gasm syntax
  - `op src, dst`
  - `%register` `$literal` `offset(memory-reference)`

- Examples:

```
movl %eax, %edx       →   edx = eax
movl $0x123, %edx     →   edx = 0x123
movl (%ebx), %edx     →   edx = *((int32_t)*) ebx)
movl 4(%ebx), %edx    →
```

Brief review of x86 assembly

- We're going to use AT&T /gasm syntax
  - `op src, dst`
  - `%register` `$literal` `offset(memory-reference)`

- Examples:

```
movl %eax, %edx        →   edx = eax
movl $0x123, %edx      →   edx = 0x123
movl (%ebx), %edx      →   edx = *((int32_t)*) ebx)
movl 4(%ebx), %edx     →   edx = *((int32_t)*) ebx+4)
```

Brief review of stack instructions

Stack operation		**equivalent**
pushl %eax	→	subl $4, %esp movl %eax, (%esp)

Brief review of stack instructions

Stack operation		**equivalent**
pushl %eax	→	subl $4, %esp movl %eax, (%esp)
popl %eax	→	movl (%esp), %eax addl $4, %esp

Brief review of stack instructions

Stack operation		**equivalent**
pushl %eax	→	subl $4, %esp movl %eax, (%esp)
popl %eax	→	movl (%esp), %eax addl $4, %esp
call $0x12345	→	pushl %eip movl $0x12345, %eip

Brief review of stack instructions

Stack operation		**Pseudo-asm equ**
pushl %eax	→	subl $4, %esp movl %eax, (%esp)
popl %eax	→	movl (%esp), %eax addl $4, %esp
call $0x12345	→	pushl %eip movl $0x12345, %eip
ret	→	popl %eip

Brief review of stack instructions

Stack operation		Pseudo-asm equ
pushl %eax	→	subl $4, %esp movl %eax, (%esp)
popl %eax	→	movl (%esp), %eax addl $4, %esp
call $0x12345	→	pushl %eip movl $0x12345, %eip
ret	→	popl %eip
leave	→	movl %ebp, %esp pop %ebp

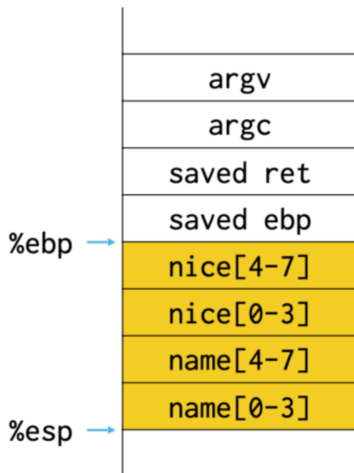
Example 0

```
int check_authentication (char *password){
int auth_flag = 0;
char password_buffer[16];

strcpy(password_buffer, password);

if (strcmp(password_buffer, "brillig")==0)
auth_flag = 1;

if (strcmp(password_buffer, "outgrabe")==0)
auth_flag = 1;

return auth_flag; }
```

passwordbuf.c

Example 1

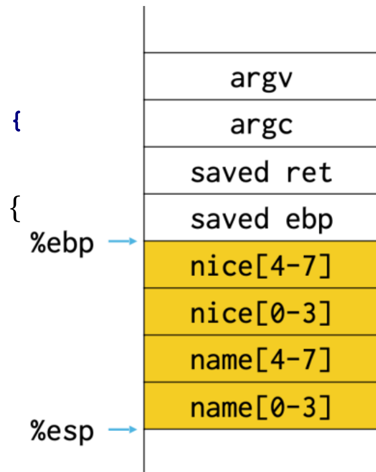
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv)
{

char nice[] = "is nice.";
char name[8];

strcpy(name,argv[1]);

printf("%s %s\n",name,nice);
return 0;

}
```



example1.c

Example 1

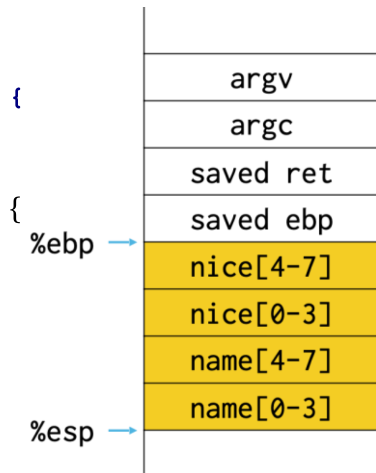
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv) {

char nice[] = "is nice.";
char name[8];

strcpy(name,argv[1]);

printf("%s %s\n",name,nice);
return 0;

}
```



What happens if we read a long name?

Example 1

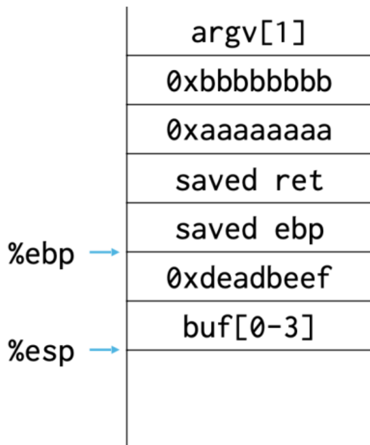
```
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv) {

char nice[] = "is nice.";
char name[8];

strcpy(name,argv[1]);

printf("%s %s\n",name,nice);
return 0;

}
```



What happens if we read a long name?
If not null terminated, can read more of the stack.

Example 2

```c
#include <stdio.h>
#include <string.h>

void foo() {

printf("hello all!!\n");
exit(0);
}

void func(int a, int b, char *str)
{
int c = 0xdeadbeef;
char buf[4];
strcpy(buf,str);
}
int main(int argc, char**argv) {
func(0xaaaaaaaa,0xbbbbbbbb,argv[1]
);
return 0;
}
```



example2.c

If program argument is long...

If program argument is long...

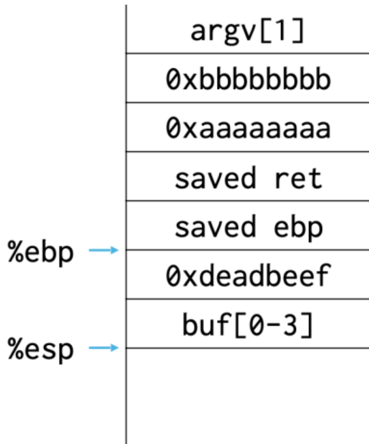
Stack buffer overflow

- If source string of `strcpy` controlled by attacker and destination on the stack:
  - Attacker gets to control where the function returns by overwriting the return address
  - Attacker gets to transfer control to anywhere
- Where do you jump?

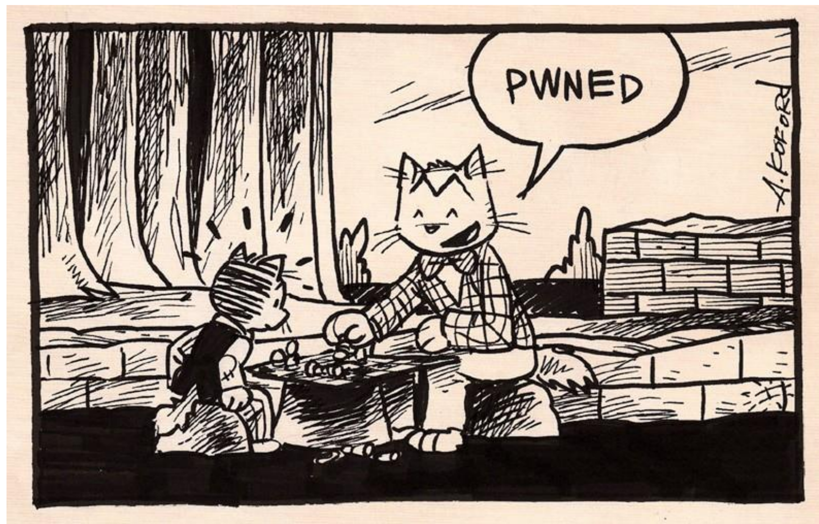
Can jump to existing functions

Overwrite saved ret with &foo.

```
#include <stdio.h>
#include <string.h>

void foo() {

printf("hello all!!\n");
exit(0);
}

void func(int a, int b, char *str)
{
int c = 0xdeadbeef;
char buf[4];
strcpy(buf,str);
}
int main(int argc, char**argv) {
func(0xaaaaaaaa,0xbbbbbbbb,argv[1]
);
return 0;
}
```

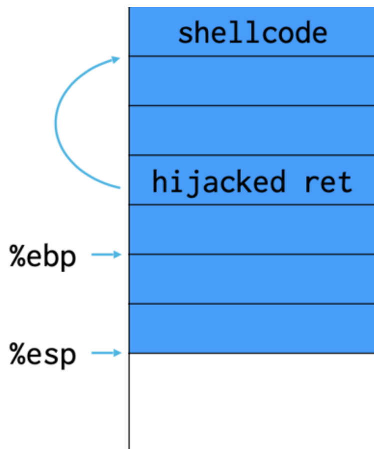


Jump to existing functions



Jump to attacker-supplied code

- Put code in string
- Jump to start of string



Shellcode

- Shellcode: Small code fragment that receives initial control in a control flow hijack exploit
- Control flow hijack: taking control of instruction pointer
- Earliest attacks used shellcode to exec a shell
- Target a setuid root program, gets you root shell

Shellcode

```
int main(void) {

char* name[1];
name[0] = “/bin/sh“;
name[1] = NULL;
execve(name[0], name, NULL);
return 0;

}
```

Can we just take output from gcc/clang?

Shellcode

- Shellcode cannot contain null characters '\0'
  - Why?
- If payload is via `gets()` must also avoid line breaks
  - Why?
- Fix: Use different instructions and NOPs.

Payload is not always robust

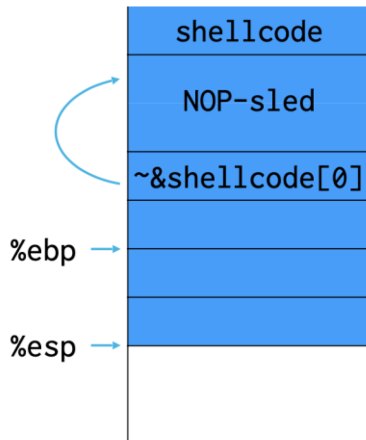
- Exact address of shellcode start not always easy to guess.

Payload is not always robust

- Exact address of shellcode start not always easy to guess.
    - A miss will result in a segfault.

Payload is not always robust

- Exact address of shellcode start not always easy to guess.
    - A miss will result in a segfault.
    - Fix: NOP sled. Fill space with NOP instructions to allow error in stack locations.



Possible Mitigations

Avoiding Buffer Overflows Attack

- Do memory auditing by using **Valgrind** memcheck

```c
#include <stdlib.h>
int main() {

int *buf = malloc(sizeof(int) * 20);
for (int i=0; i<20; i++){
buf[i] = i;

// forgot to free buf
}
 }
```

valgrind.c

```
==557==          suppressed: 0 bytes in 0 blocks
==557== Rerun with --leak-check=full to see details of leaked memory
==557==
==557== For counts of detected and suppressed errors, rerun with: -v
==557== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
rabeshi@LAPTOP-2MBC0JI1:~$ valgrind

Command 'valgrind' not found, but can be installed with:

sudo apt install valgrind
```

Avoiding Buffer Overflows Attack

- Use fgets instead of gets()
- Use strncmp() instead of strcmp()
- Use strncpy() instead of strcpy()

Use of gets

```
#include<stdio.h>
int main() {
char string[10];

printf("Enter the String: ");
gets(string);

printf("\n%s",string);
return 0;

}
```

Use of fgets

```
#include<stdio.h>
int main() {
char string[20];
printf("Enter the string:
");

fgets(string,20,stdin);
#input from stdin stream

printf("\nThe string is:
%s",string);
return 0;
}
```

Group Task

Question 1

Where can an attacker who is trying to “smash the stack” put their attack code if the buffer to be overflowed is on the stack?

a. On the stack before the return pointer
b. On the stack frame of another function
c. On the heap
d. All of the above

Question 2

Which of these kinds of buffer overflows can be a security threat?

a. Unsafe Library function calls
b. Buffers that store internal data
c. Stack smashing
d. All of the above

Question 3

Use **Valgrind** to uncover the problem with the below.

```
int fun(int n) {
char *pInfo = malloc (n *sizeof(char));

/* Do some work */

return 0;
}
```

Next up: Defenses and more advanced attacks.