Vulnerabilities in C/C++ programs – Part I

TDDC90 – Software Security

Ulf Kargén

Department of Computer and Information Science (IDA)

Division for Database and Information Techniques (ADIT)

The C and C++ languages

- Programs compile directly to machine code
- Fine-grained control of memory given to programmers
  - ⇨ Optimized for speed – not reliability
  - ⇨ Subtle mistakes can have devastating security implications!
  - ⇨ Understanding of low-level details necessary to take full advantage of language, and to avoid introducing vulnerabilities
    - Easy to make mistakes when coming from e.g. C# or Java!

Outline of lectures

First lecture

- Introduction and motivation
- Assembly language primer
- Vulnerabilities and exploits

Second lecture

- More vulnerabilities and exploits
- Writing secure code
- Mitigations
- “Modern” exploit techniques

Introduction and motivation

Why look at vulnerabilities in C/C++ code?

C and C++ are old languages with known security problems

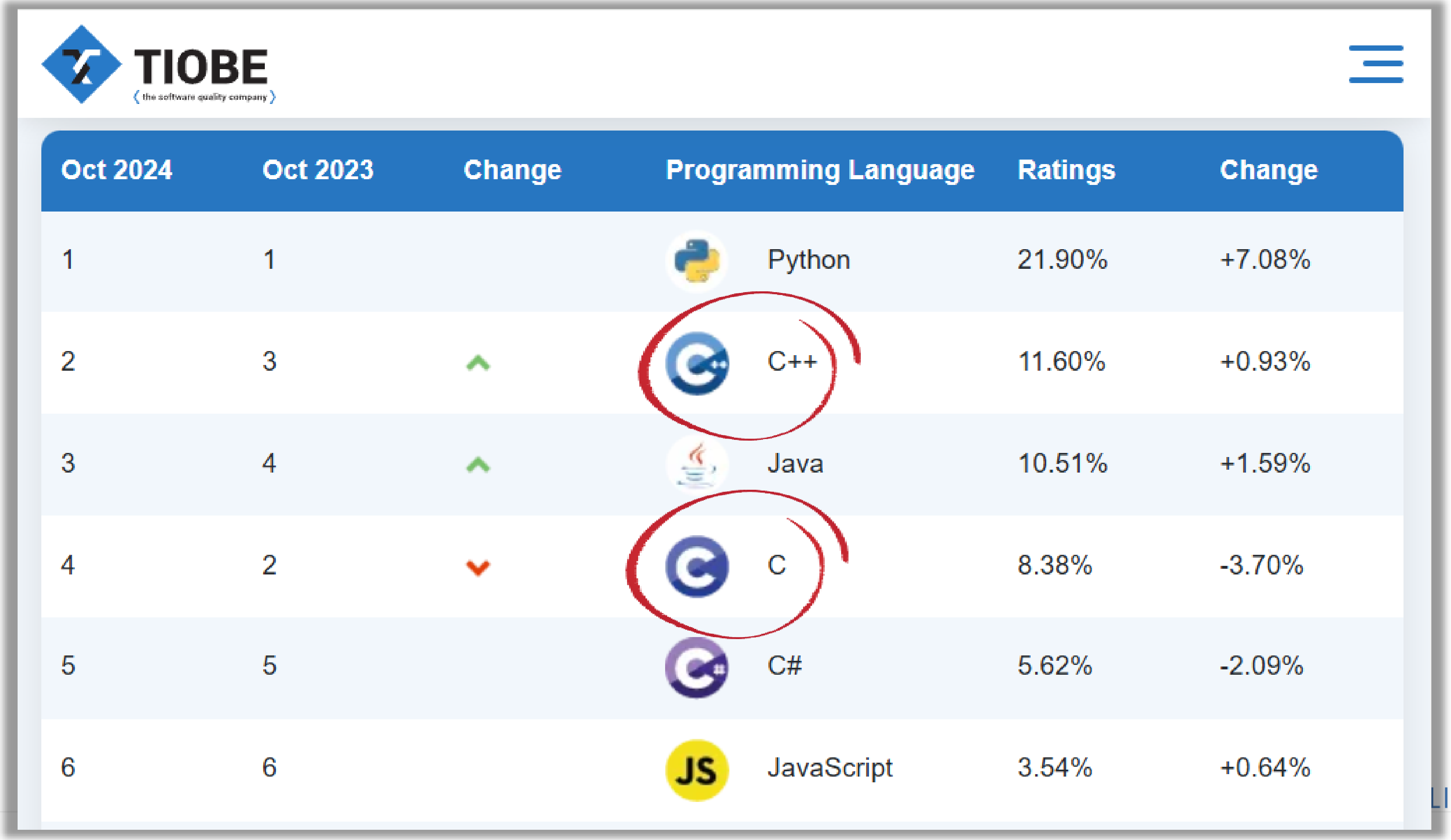
⇨ Why not just implement everything in Java / C# / Python / Rust and be done with it?

- Some code need to run “close to the metal” (OS kernels, device drivers)
- Performance reasons:
  - Low-powered devices (little RAM, slow CPU) e.g., IoT devices

- **Huge amounts of code already written in C/C++ that need to be maintained for decades to come…**

Why look at vulnerabilities in C/C++ code?

TIOBE language popularity index, October 2024:



Oct 2024	Oct 2023	Change	Programming Language	Ratings	Change
1	1		Python	21.90%	+7.08%
2	3	▲	C++	11.60%	+0.93%
3	4	▲	Java	10.51%	+1.59%
4	2	▼	C	8.38%	-3.70%
5	5		C#	5.62%	-2.09%
6	6		JavaScript	3.54%	+0.64%

Why study attack techniques?

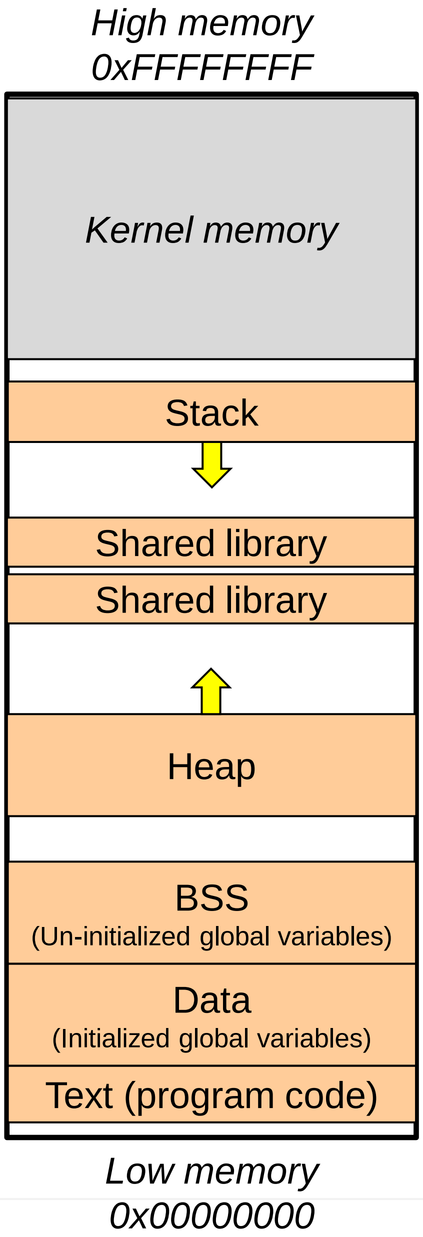
- “Know thy enemy”
  - How could you possibly protect from attacks if you don’t know what techniques attackers use?
- Important to be able to tell if a bug has security implications
  - Scheduling/prioritizing patches
  - Decide what to publish on e.g. public bug trackers

Assembly language primer

Linux memory layout and x86 basics

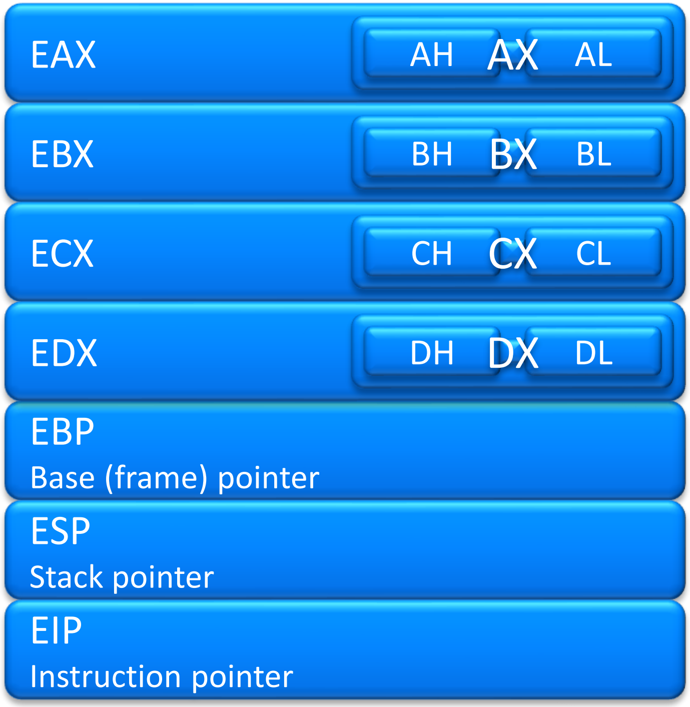
Memory layout of x86 Linux

(What you will use in the Pong lab)



- All processes see 4GB of private continuous virtual memory. (Mapped by OS to RAM)
- The **stack** is located at high memory addresses and **grows downwards in memory**
  - Used for storing local variables of function calls, function call parameters, return addresses, etc.
  - An element on the stack is always 32 bits (4 bytes) (for example, pushing a single byte to the stack requires that it is first zero or sign extended to 4 bytes)
- Main executable (Text), and its Data and BSS segment, is located in low memory
- The **heap** is located above the Text, Data, and BSS segment. **Grows upwards in memory**.
  - Used for dynamically allocated memory (*malloc*, *new*)
- Note: x86 is a *little-endian* architecture: First byte of e.g. a 4-byte word is the *least* significant byte.

Registers on the x86



Additional registers

- ESI and EDI
- CS, SS, DS, ES, FS, GS
- EFLAGS
- …

- Four general-purpose 4-byte registers (EAX - EDX)
- Partial registers
  - 2 least significant bytes of full register (***n***X)
  - Bytes 1 and 2 of ***n***X called respectively ***n***L and ***n***H (*L*ow and *H*igh)

Special registers

- ESP – points to topmost element of stack
- EBP – points to current *frame* (on the stack), which contains local variables of one function call. Local variables accessed relative to EBP.
- EIP – points to the currently executing instruction

Assembly language mnemonics

Intel style

- *opcode destination*, *source*
- mov [esp+4], eax

AT&T (gcc, gdb) style

- *opcode source*, *destination*
- movl %eax, 4(%esp)

mov *dst*, *src*	Copy the data in *src* to *dst*
add/sub *dst*, *src*	Add/subtract the data in *src* to the data in *dst*
and/xor *dst*, *src*	Bitwise AND/XOR the data in *src* with the data in *dst* and store result in *dst*
push *target*	Push *target* onto the stack, decrementing ESP
pop *target*	Pop *target* from the stack, incrementing ESP
lea *dst*, *src*	Load the address of *src* into *dst*
call *address*	Push address of the next instruction onto stack and set EIP to *address*
ret	Pop EIP from the stack
leave	Exit a high-level function (copy EPB to ESP, pop EBP from stack)
jcc address	Jump to *address* if condition code *cc* (e.g. e, ne, ge) is set
jmp *address*	Jump to *address*
int *value*	Call interrupt of *value* (0x80 will perform a Linux system call)

Semantics of some important x86 instructions

- **push** <*op*>
  Equivalent to:
  esp = esp – 4
  [esp] = <*op*>

Access memory *pointed to* by esp

- **pop** <*op*>
  Equivalent to:
  <*op*> = [esp]
  esp = esp + 4

- **call** <*function address*>
  Instruction for performing a function call. Pushes return address to stack and jumps to start of called function.
  Equivalent to:
  push <*address of next instruction*>
  eip = <*function address*>

- **ret**
  Used to return from function. Pops return address from stack and jumps back to the calling function.
  Equivalent to:
  pop eip

Direct vs indirect branches

Direct branches

Addresses are **hardcoded** offsets relative to current instruction pointer

Examples:

- **call** 0x123
  Equivalent to:
  push <address of next instruction>
  eip = eip + 0x123 (291 decimal)
- **jmp** 0x123
  Equivalent to:
  eip = eip + 0x123
- ***jcc*** 0x123
  Conditional branches are always direct

Indirect branches

Addresses are **stored in a register or memory,** i.e. decided at **runtime**

Examples:

- **call** eax
  Equivalent to:
  push <address of next instruction>
  eip = eax
- **jmp** eax
  Equivalent to:
  eip = eax
- **ret**
  Target address stored on stack

Used to implement calls via function pointers

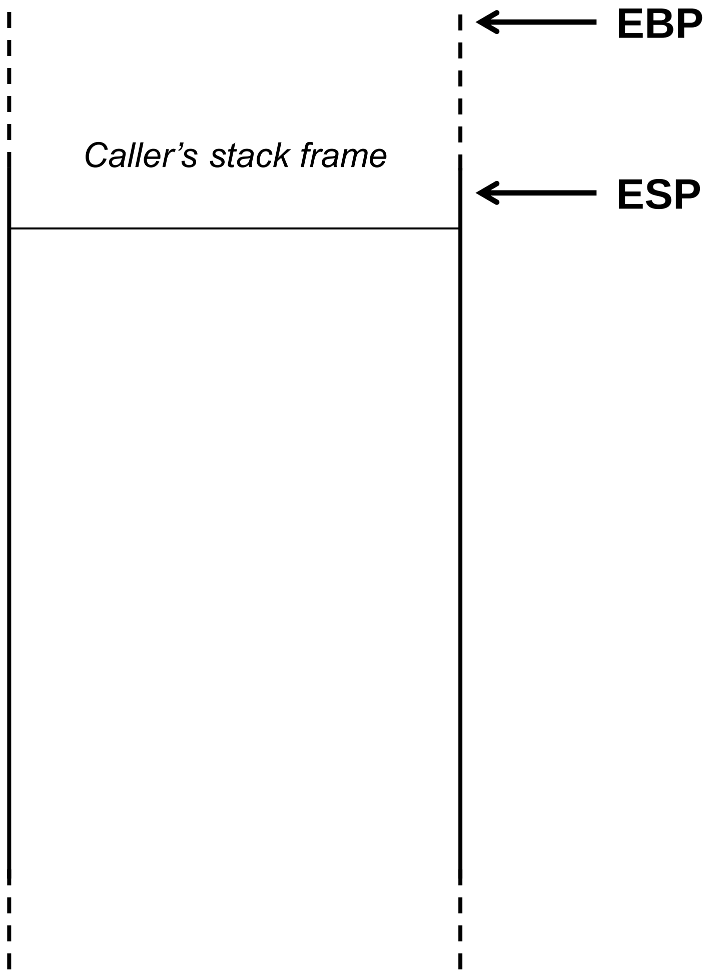
Function calls on x86 (stdcall)

1. Caller pushes arguments from right to left onto stack
2. Caller issues a ‘call’ instruction – pushes return address and jumps to function start.
3. Function *prologue* executes
    a. Pushes old value of EBP to stack, updates EBP to point to saved EBP on stack
    b. Subtracts ESP to allocate space for local variables
4. Function main logic executes
5. Function *epilogue* executes
    a. Puts return value (if any) into EAX register
    b. “Deallocates” local variables on stack by increasing ESP
    c. Pops saved EBP into EBP
    d. Issues a ‘ret’ instruction – pops return address of stack and jumps to that address
6. Caller removes arguments from stack

Function calls on x86 (stdcall)

Example

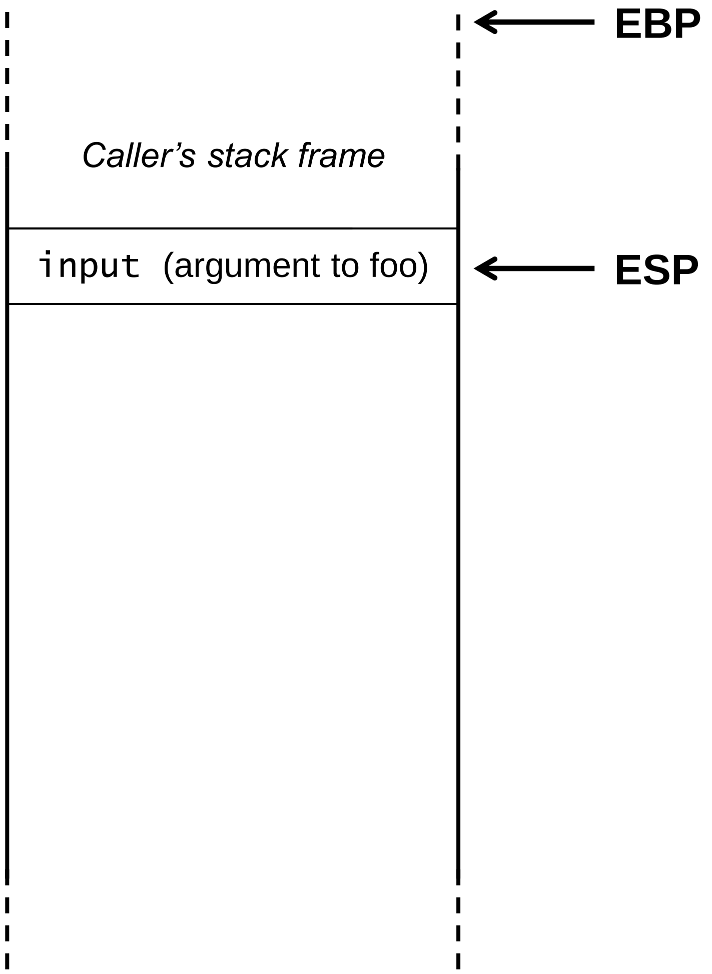
```
.
.
   foo(user_data);
.
.

void foo(char* input)
{
   unsigned int len;
   char buffer[16];

   len = strlen(input);
   strcpy(buffer, input);

   printf(“%s: %d\n”, buffer, len);
}
```



Function calls on x86 (stdcall)

Example

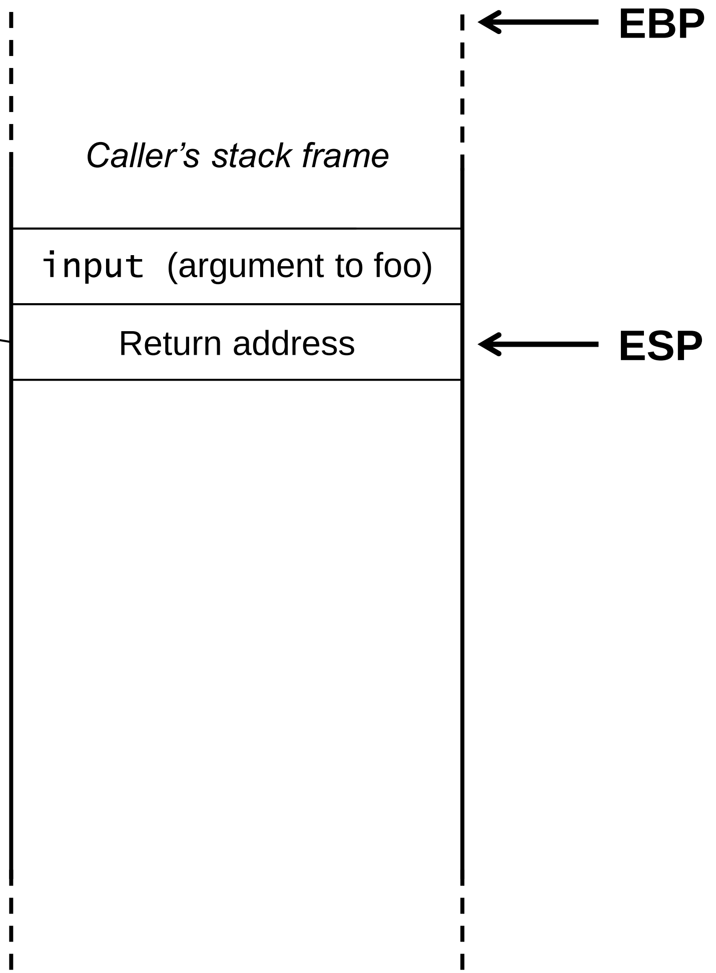
```
.
.
   foo(user_data);
.
.

void foo(char* input)
{
   unsigned int len;
   char buffer[16];

   len = strlen(input);
   strcpy(buffer, input);

   printf(“%s: %d\n”, buffer, len);
}
```



Function calls on x86 (stdcall)

Example

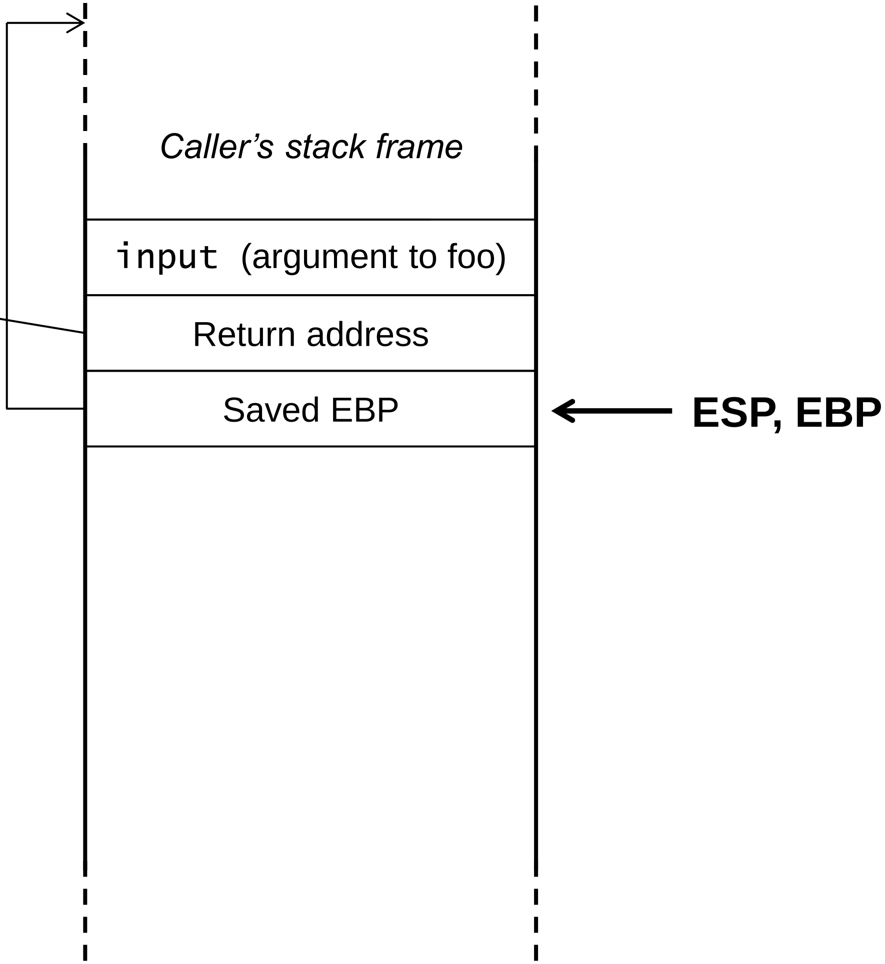
```
.
.
   foo(user_data);
.
.

void foo(char* input)
{
   unsigned int len;
   char buffer[16];

   len = strlen(input);
   strcpy(buffer, input);

   printf("%s: %d\n", buffer, len);
}
```



Function calls on x86 (stdcall)

Example

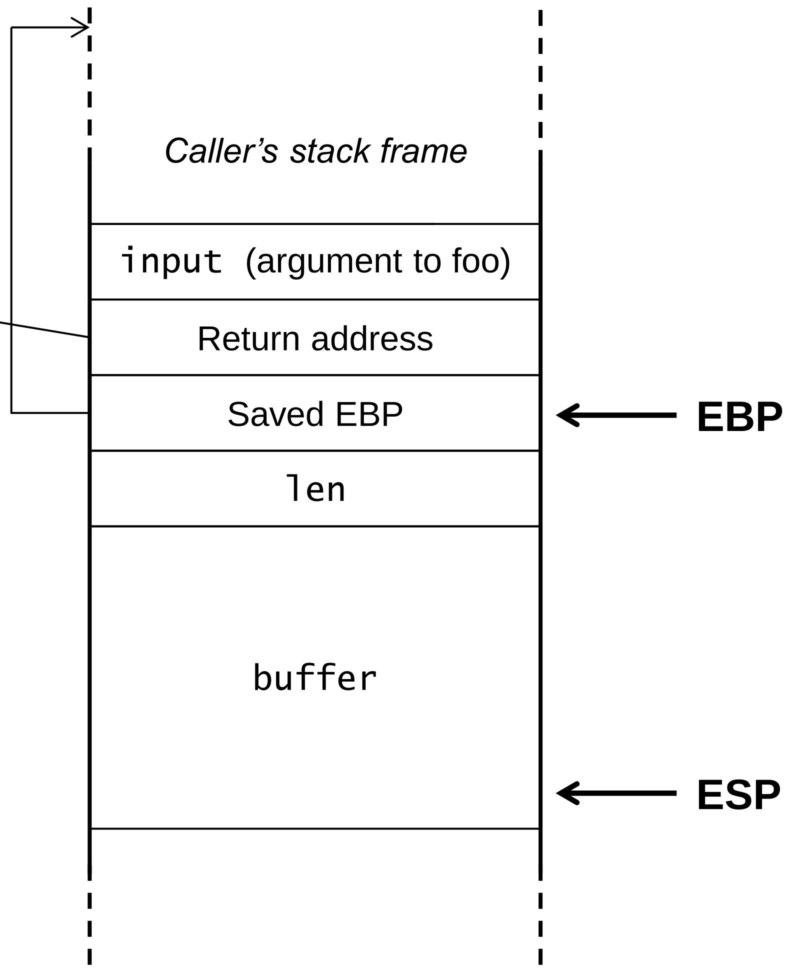
```
.
.
   foo(user_data);
.
.

void foo(char* input)
{
   unsigned int len;
   char buffer[16];

   len = strlen(input);
   strcpy(buffer, input);

   printf("%s: %d\n", buffer, len);
}
```



Function calls on x86 (stdcall)

Example

```
.
.
   foo(user_data);
.
.

void foo(char* input)
{
   unsigned int len;
   char buffer[16];

   len = strlen(input);
   strcpy(buffer, input);

   printf(“%s: %d\n”, buffer, len);
}
```



Caller’s stack frame

input (argument to foo)

Return address

Saved EBP

len

buffer

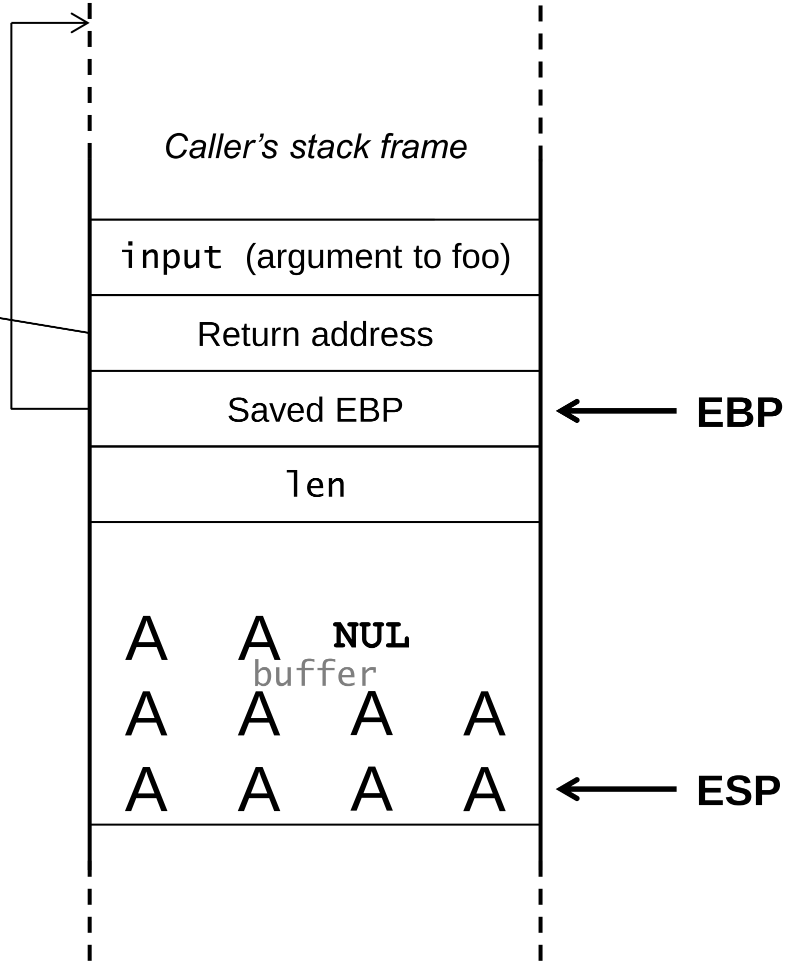
EBP

ESP

Function calls on x86 (stdcall)

Example

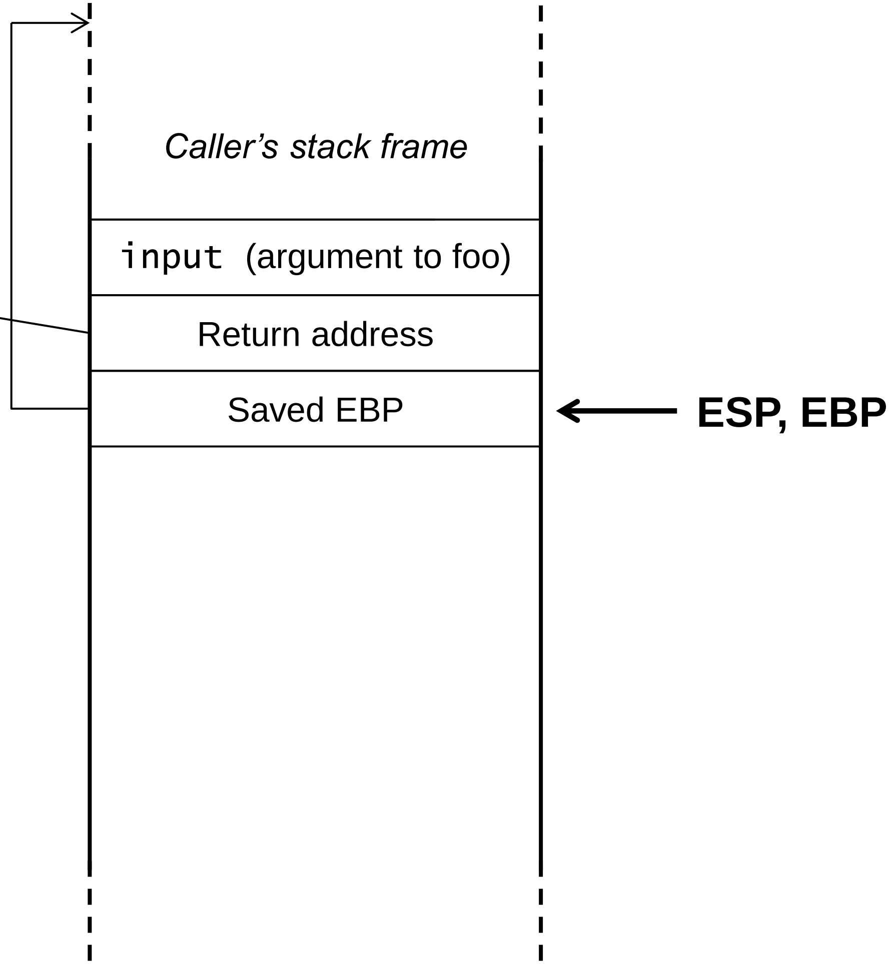
```
.
.
   foo(user_data);
.
.

void foo(char* input)
{
   unsigned int len;
   char buffer[16];

   len = strlen(input);
   strcpy(buffer, input);

   printf("%s: %d\n", buffer, len);
}
```



Function calls on x86 (stdcall)

Example

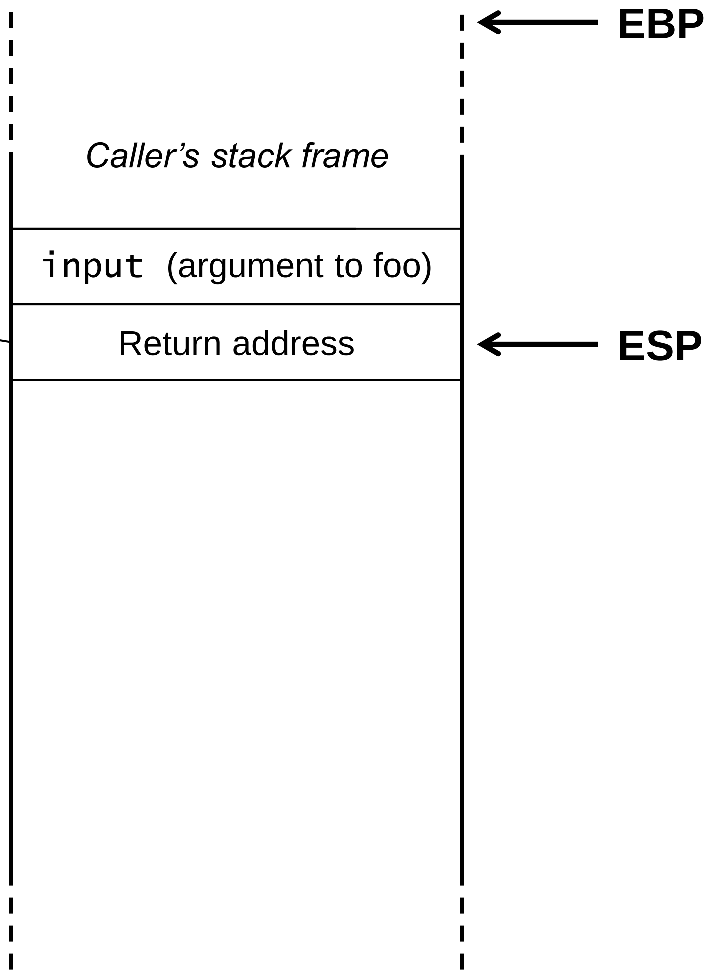
```
.
.
   foo(user_data);
.
.

void foo(char* input)
{
   unsigned int len;
   char buffer[16];

   len = strlen(input);
   strcpy(buffer, input);

   printf("%s: %d\n", buffer, len);
}
```



Function calls on x86 (stdcall)

Example

```
.
.
   foo(user_data);
.
.

void foo(char* input)
{
   unsigned int len;
   char buffer[16];

   len = strlen(input);
   strcpy(buffer, input);

   printf(“%s: %d\n”, buffer, len);
}
```



EBP

Caller’s stack frame

input (argument to foo)

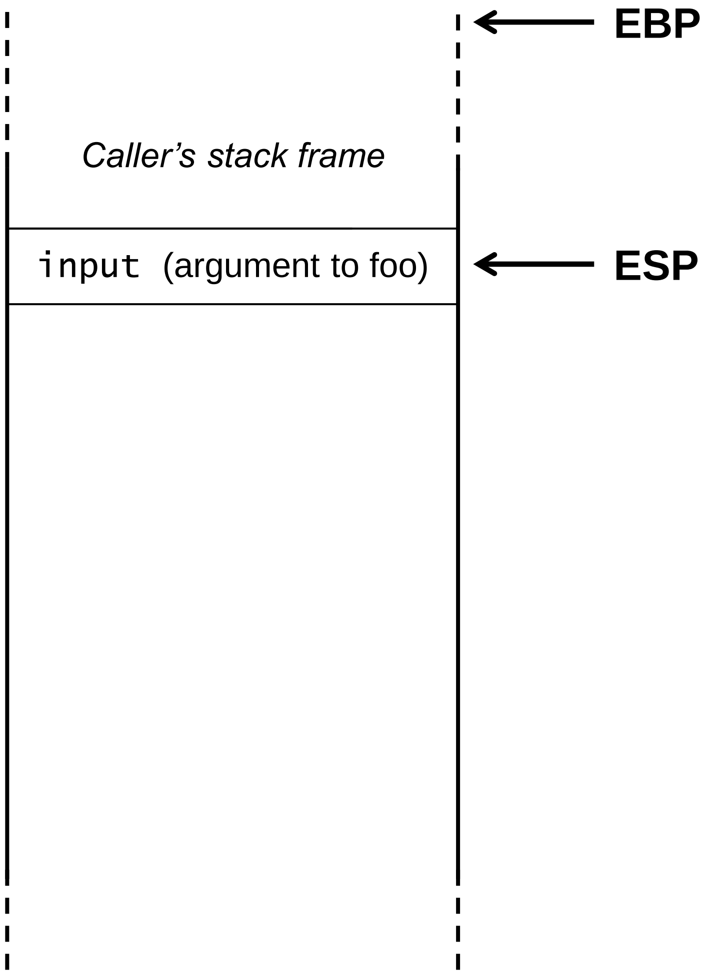
Return address

ESP

Function calls on x86 (stdcall)

Example

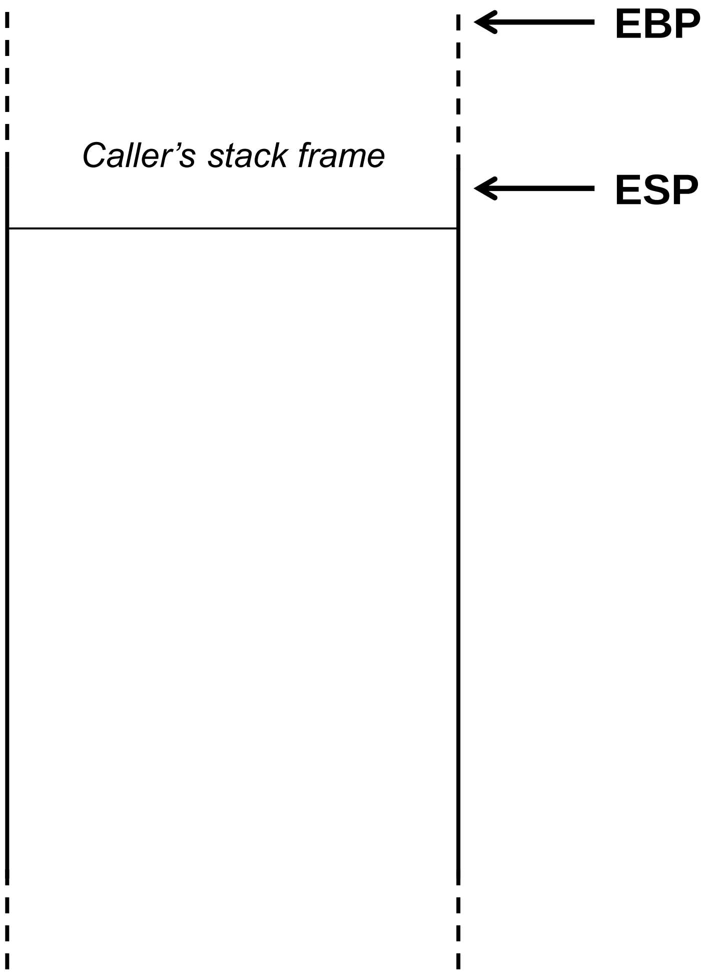
```
.
.
   foo(user_data);
.
.

void foo(char* input)
{
   unsigned int len;
   char buffer[16];

   len = strlen(input);
   strcpy(buffer, input);

   printf(“%s: %d\n”, buffer, len);
}
```



Function calls on x86 (stdcall)

Example

```
.
.
   foo(user_data);
.
.

void foo(char* input)
{
   unsigned int len;
   char buffer[16];

   len = strlen(input);
   strcpy(buffer, input);

   printf("%s: %d\n", buffer, len);
}
```



Vulnerabilities and exploits

Vulnerabilities and exploits

- Vulnerabilities
  - Flaws that makes it possible for a program to fail to meet its security requirements
- What is an exploit?
  - A verb: Exploiting a vulnerability means to take advantage of a vulnerability to compromise security.
  - A noun: An exploit is a procedure or piece of code that performs the above.
- The purpose of an exploit
  - Arbitrary code execution – Completely take over program execution to do anything the attacker wishes.
  - Information disclosure – Leak sensitive information, e.g. Heartbleed
  - Denial of Service – Disrupt functionality of a service, e.g. crash a web server
  - Privilege escalation – Gain higher privileges than what is allowed according to system policy. May be combined with arbitrary code execution exploits to completely compromise system.
    - Example: Program running as SUID root in Unix, or with Administrator/SYSTEM privileges in Windows.

Vulnerabilities and exploits

- Local and remote exploits
  - Local exploit – Physical access to system, or valid remote login credentials, required for exploit.
  - Remote exploit – “Anyone” on e.g. the Internet can perform exploit. Example: Web server exploitable by external requests.
- Severity of a vulnerability depends on what kind of exploits it enables
  - Remote exploit leading to arbitrary code execution
  - Local DoS exploit
  - Local code execution exploit without privilege escalation

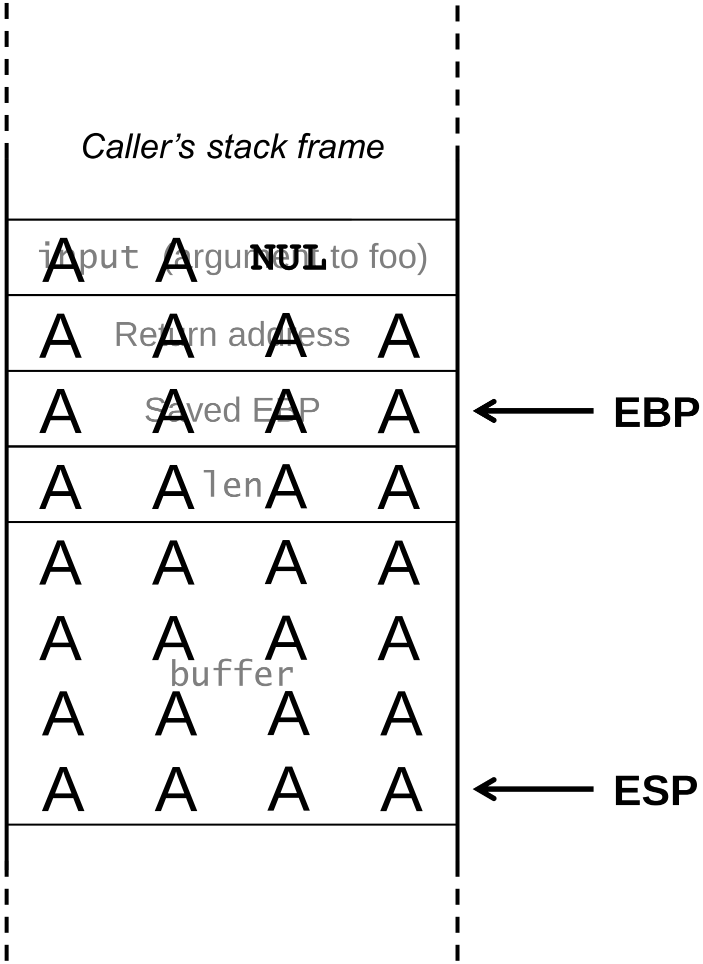
Vulnerabilities and exploits

- Local and remote exploits
  - Local exploit – Physical access to system, or valid remote login credentials, required for exploit.
  - Remote exploit – "Anyone" on e.g. the Internet can perform exploit. Example: Web server exploitable by external requests.
- Severity of a vulnerability depends on what kind of exploits it enables
  - Remote exploit leading to arbitrary code execution – Really, really bad!
  - Local DoS exploit – Not as bad?
  - Local code execution exploit without privilege escalation – Meaningless?

The “Hello World” exploit

Simple buffer overflow on the stack

```
void foo(char* input)
{
   unsigned int len;
   char buffer[16];
...
   strcpy(buffer, input);
...
```



Let’s return to our function ‘foo’ from before

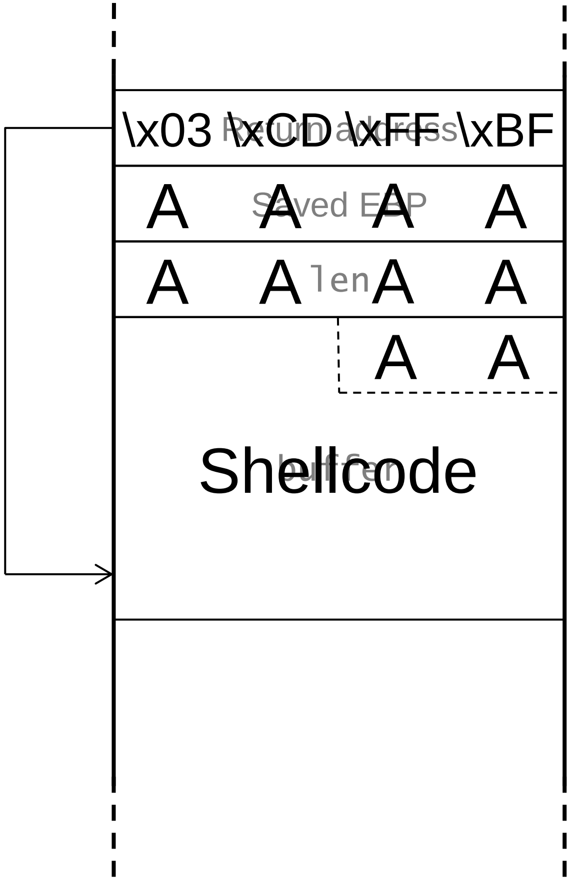
- What happens if ‘input’ is longer than 15 bytes?
- Buffer overflows, overwriting return address if string is long enough.
  - ⇨ Program later crashes when trying to return to address 0x41414141 (“AAAA”)
  - ⇨ Results in DoS. How to achieve arbitrary code execution?

The “Hello World” exploit

Arbitrary code execution

Idea: Include executable machine code in input string, and set the overwritten return pointer to point to that code.

- Such code is often referred to as “shellcode” – traditionally often used to open a command shell with elevated privileges.
- Payload consists of shellocode + padding (some A:s) + new “return” address
  - Note 1: Due to x86 being little-endian, each byte of the address (here BFFFCD03 in hex) need to be given in reverse order when crafting the string (i.e. “\x03\xCD\xFF\xBF”)
  - Note 2: Payload must usually not contain any bytes with the value zero. Recall that zero (NUL) terminates the string.
  - Note 3: This payload may not work for ‘foo’ since buffer is only 16 bytes (not enough space for code). Also possible to e.g. put shellcode before return address on stack, in the caller’s stack frame.



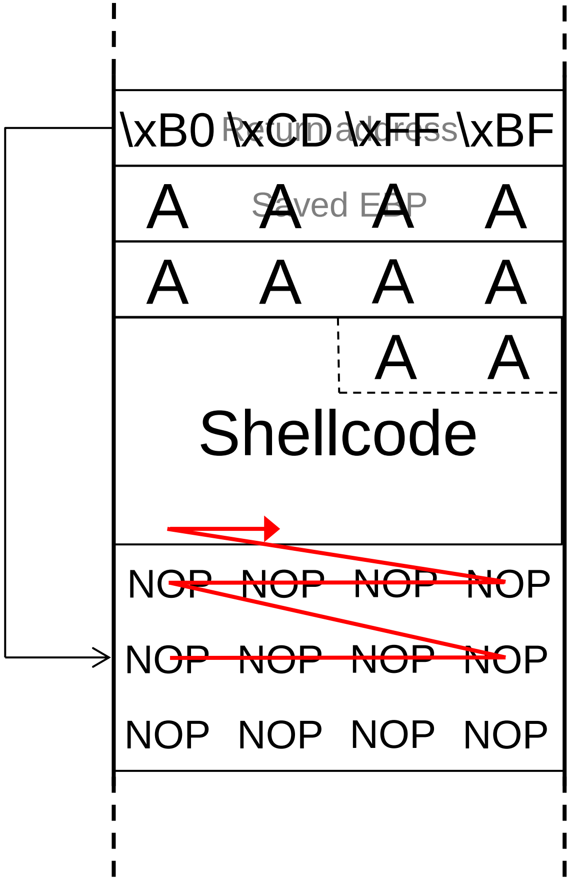
- Problem: The above approach requires that we can precisely predict absolute address of shellcode on stack.
  - Typically not possible in practice!

The “Hello World” exploit

Making the exploit reliable: Solution 1 – The NOP sled

To avoid having to know the exact shellcode address, we can use a *NOP sled*

- Precede the shellcode with a sequence of NOP instructions.
  - A NOP instruction (hex \x90) does nothing, except of advancing the instruction counter one byte.
- Point the return address somewhere in the middle of the NOP sled
- Gives some “wiggle room” – As long as the return address points somewhere into the NOP sled, execution will follow the NOPs into the shellcode.
- Drawbacks:
  - Requires larger buffers
  - Still need to know approximate address of NOP sled

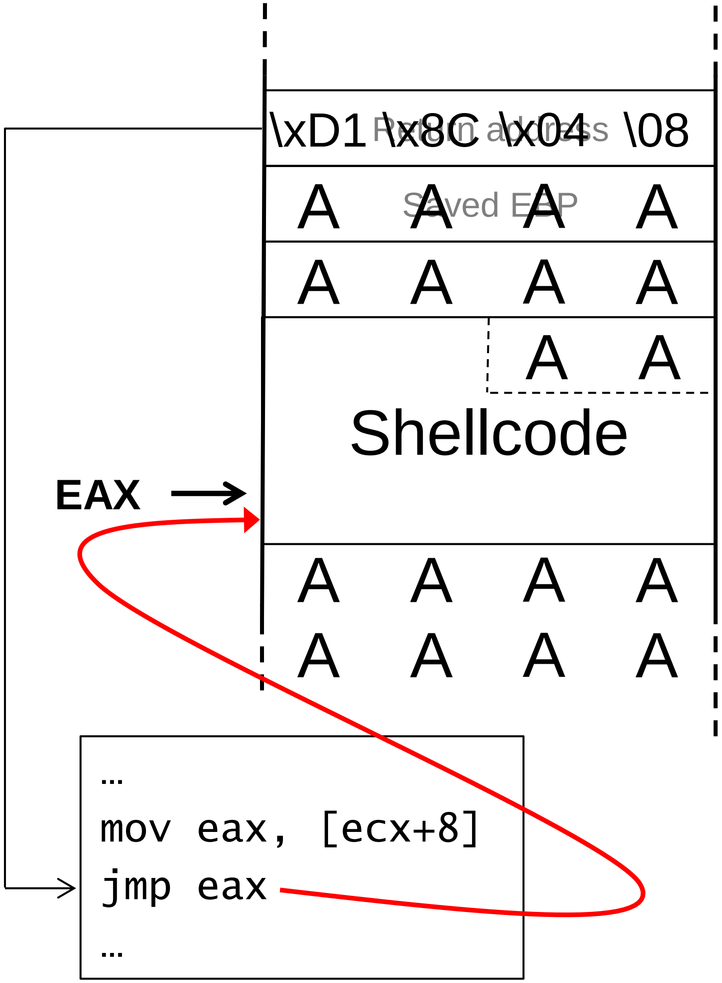


The “Hello World” exploit

Making the exploit reliable: Solution 2 – Register trampolines

A more robust solution than the NOP sled is to use *register trampolines* (a.k.a. *register springs*)

- Find a register *REG* that *right before the function returns* points to data that you control.
  - Given that function behavior is deterministic, if *REG* points to data on the stack, it will always point to the same location *relative to the beginning of the stack frame*.
- Make sure your shellcode starts at just the location pointed to by *REG*
- Find an instruction in an executable image (main executable or shared library) that performs an indirect jump to address in *REG*
- Overwrite return address with the address *to the jump instruction*.
- When function “returns”, it will jump to the instruction, which in turn will jump to the shellcode.
- Obviously not always possible to find suitable *REG* and jump instruction.



```
…
mov eax, [ecx+8]
jmp eax
…
```

Stack-buffer overflow variations

The function may alter parts of the overwritten stack area prior to returning – Special “tricks” often needed in practice

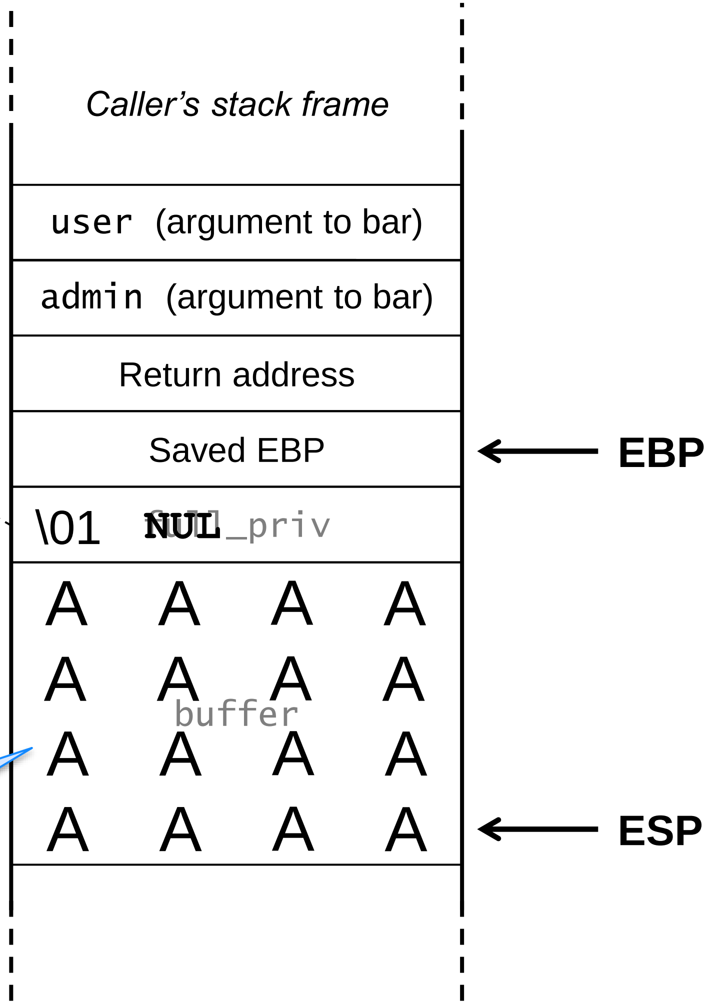
- Insert code that jumps past altered parts of stack to shellcode
- Put shellcode in environment variables
- Put shellcode in other buffers (e.g. on heap)
- …

If return address cannot be overwritten, other targets are also possible

- Overwrite saved EBP – alters stack frame of *calling* function
- Overwrite function pointers on stack
- Overwrite other sensitive non-control data (i.e. data that is not a pointer to code)

Data-only attacks
Example

```
void bar(char* user, bool isAdmin)
{
   bool full_priv = isAdmin;
   char buffer[16];

   strcpy(buffer, user);

   if(full_priv)
      // Do privileged stuff
   else
      printf(
         “User %s is not admin \n”,
         user);
}
```



Overwrite full_priv
(to any non-zero value) with
buffer overflow

Special case: Off-by-one errors

Special case of stack-based overflows where only a single byte can be written past buffer bounds – Often more subtle than “regular” buffer overflows.

Example:

```
char buffer[100];
if(strlen(input) > 100)
{
    printf(“String too long!”);
    exit(1);
}
strcpy(buffer, input);
```

Should be:

```
char buffer[100];
if(strlen(input) >= 100)
{
    printf(“String too long!”);
    exit(1);
}
strcpy(buffer, input);
```

Is this safe?

- No! ‘strlen’ does not include the space needed for the NULL-terminator.
- Using a 100-character string results in a NULL-byte being written past end of buffer.
- Could e.g. overwrite least significant byte of saved EBP to alter context of calling function. Can lead to arbitrary code execution!

Examples of stack-based buffer overflows

Real-life overflow in FTP server

```
char mapped_path[MAXPATHLEN];
if(!(mapped_path[0] == '/' && mapped_path[1] == '\0'))
    strcat(mapped_path, "/");
strcat(mapped_path, dir);
```

Real-life overflow in web server (the pointer ‘ptr’ points to user-controllable data)

```
int resolve_request_filename(char *ptr)
{
  char filename[255];
  ...
  if(!strncmp(ptr, thehost->CGIDIR, strlen(thehost->CGIDIR))) {
    strcpy(filename, thehost->CGIROOT);
    ptr += strlen(thehost->CGIDIR);
    strcat(filename, ptr);
  } else {
    strcpy(filename, thehost->DOCUMENTROOT);
    strcat(filename, ptr);
    ...
```

Examples of stack-based buffer overflows

A more subtle example

Off-by-one overflow in the wu-ftpd FTP server

```
/*
 * Join the two strings together, ensuring that the right thing
 * happens if last component is empty, or the dirname is root.
 */

if (resolved[0] == '/' && resolved[1] == '\0')
    rootd = 1;
else
    rootd = 0;

if (*wbuf) {
    if (strlen(resolved) + strlen(wbuf) + rootd + 1 > MAXPATHLEN) {
        errno = ENAMETOOLONG;
        goto err1;
    }
    if (rootd == 0)
        (void) strcat(resolved, "/");
    (void) strcat(resolved, wbuf);
```

Avoiding buffer overflows

Some best practices

- Perform input validation
  - Never trust user-supplied data!
  - Accept only “known good” instead of using a blacklist
  - Always perform correct bounds-checking before copying data to buffers
- Use safe(r) APIs for string operations
  - E.g. `strncpy(dst, src, len)` instead of `strcpy(dst, src)`
  - Beware: `strncpy` (and `strncat`) don’t NULL terminate strings if the length of ‘`src`‘ is larger than or equal to the maximum allowed (i.e. >= ‘`len`’)
  - The following code leads to information leakage if `strlen(str) >= 100` (Stack content beyond ‘`buffer`‘ is printed, until a zero-byte is encountered) – Can also lead to code execution under some conditions.

```
char buffer[100];
strncpy(buffer, str, sizeof(buffer));
...
printf(“%s”, buffer);
```

Avoiding buffer overflows

Some best practices

- Make sure to terminate strings when using the strn-functions.

```
char buffer[100];
strncpy(buffer, str, sizeof(buffer));
buffer[sizeof(buffer) – 1] = 0;
...
printf(“%s”, buffer);
```

- Use `strlcpy`, `strlcat` where available. These guarantee correct string termination.
  - Note: These are not part of the standard C library. Not available on many systems (including the one you use for Pong).

Heap-based buffer overflows

- Often similar causes as stack-based buffer overflows

- Also often exploitable, but different methods compared to overflows on the stack (no return pointer to overwrite)
  - Overwrite function pointers or C++ VTable entries in other heap-allocated objects
  - Overwrite memory allocator metadata

Heap-based buffer overflows

Chunks of memory allocated on the heap are often adjacent to each other – Overflowing from one chunk into another possible

- Possible to gain control by overflowing a heap-allocated buffer and overwriting function pointers in adjacent object on heap.
- Use e.g. one of previously discussed methods to “find” shellcode in memory
  - (Semi)predicable location on stack or heap + NOP sled
  - Register trampolines
  - Shell code in environment variable, etc.

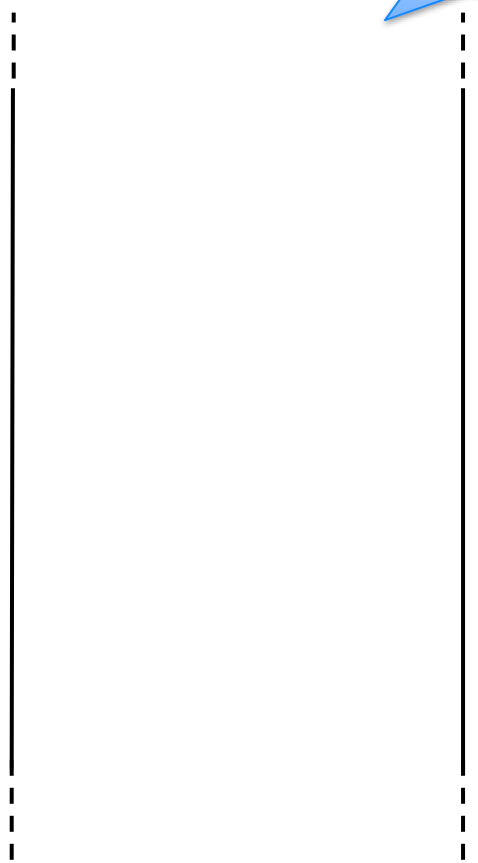
Function pointer overwrites

Example (C++)

```
struct MyStruct {
   int var_a;
   void (*fun_ptr)(int);
};

void fun(int i);

void baz(char* str) {
   MyStruct* s = new MyStruct;
   s->fun_ptr = &fun;

   char* buffer = new char[16];
   strcpy(buffer, str);

   s->fun_ptr(5);
```

Function pointer

Heap (not stack)

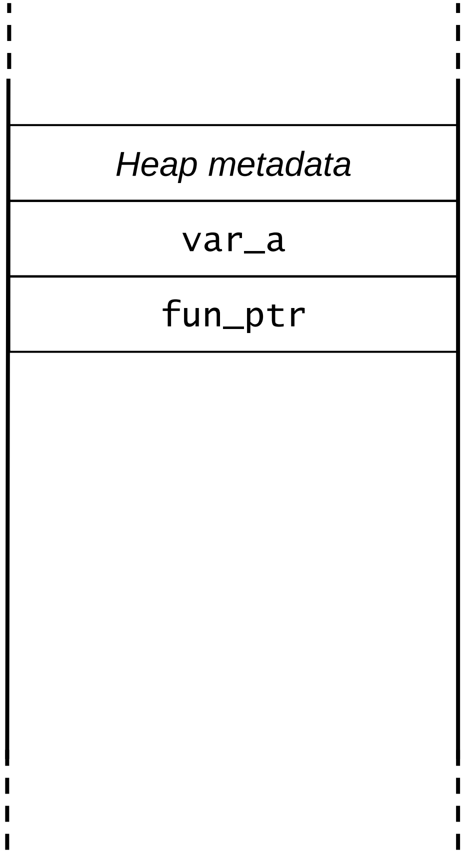


Function pointer overwrites

Example (C++)

```
struct MyStruct {
   int var_a;
   void (*fun_ptr)(in
};

void fun(int i);

void baz(char* str) {
   MyStruct* s = new MyStruct;
   s->fun_ptr = &fun;

   char* buffer = new char[16];
   strcpy(buffer, str);

   s->fun_ptr(5);
```

Allocate new `MyStruct` object

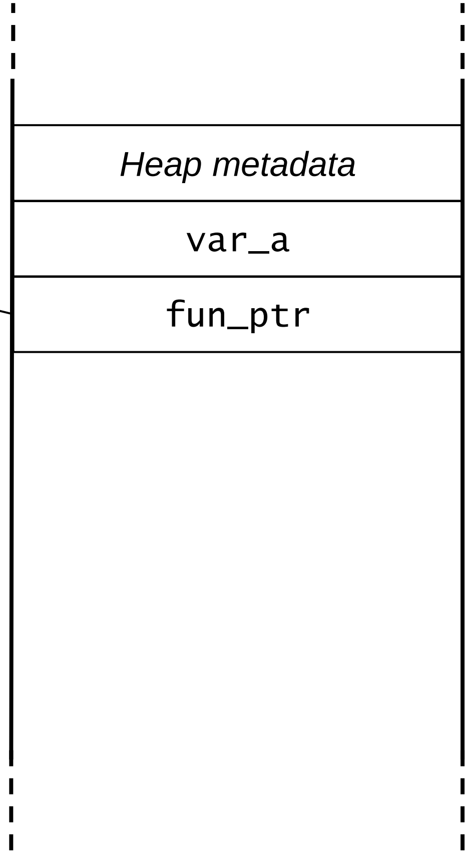


Function pointer overwrites

Example (C++)

```
struct MyStruct {
   int var_a;
   void (*fun_ptr)(int);
};

void fun(int i);

void baz(char* str) {
   MyStruct* s = new MyStruct;
   s->fun_ptr = &fun;

   char* buffer = new char[16];
   strcpy(buffer, str);

   s->fun_ptr(5);
```

Set fun_ptr to point to fun

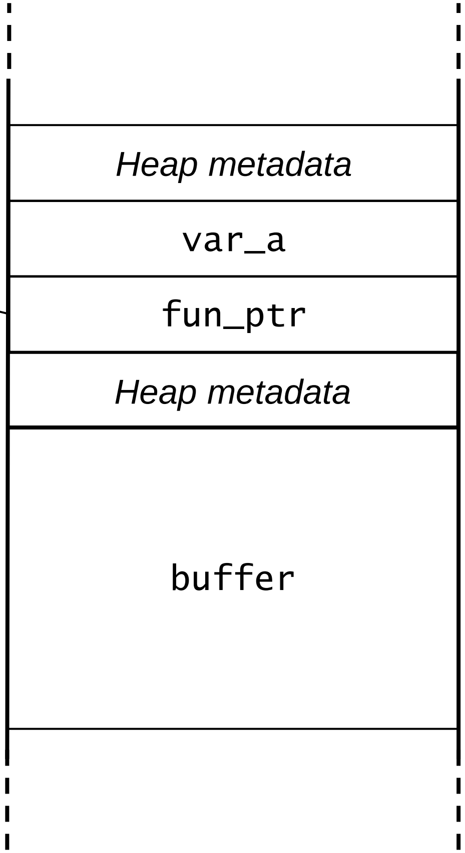


Function pointer overwrites

Example (C++)

```
struct MyStruct {
   int var_a;
   void (*fun_ptr)(int);
};

void fun(int i);

void baz(char* str) {
   MyStruct* s = new MyStruct;
   s->fun_ptr = &fun;

   char* buffer = new char[16];
   strcpy(buffer, str);

   s->fun_ptr(5);
```

Allocate 16-byte buffer

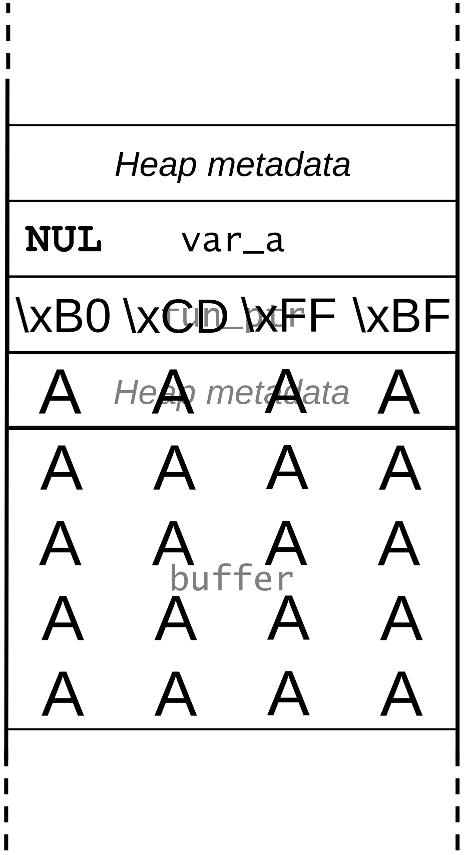


Function pointer overwrites

Example (C++)

```
struct MyStruct {
   int var_a;
   void (*fun_ptr)(int);
};

void fun(int i);

void baz(char* str) {
   MyStruct* s = new MyStruct;
   s->fun_ptr = &fun;

   char* buffer = new char[16];
   strcpy(buffer, str);

   s->fun_ptr(5);
```

Buffer overflow!



Heap metadata

NUL var_a

\xB0 \xCD \xFF \xBF

fun_ptr

A A A A

Heap metadata

A A A A

A A A A

A A A A

A A A A

buffer

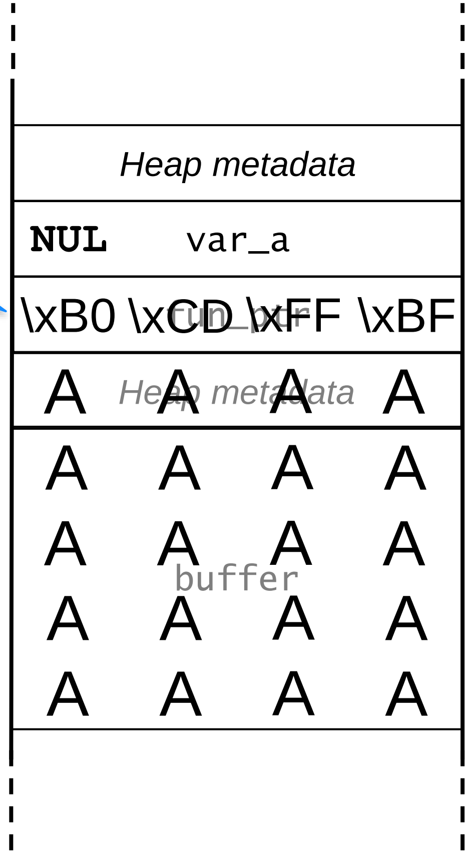
Function pointer overwrites

Example (C++)

```
struct MyStruct {
   int var_a;
   void (*fun_ptr)(int);
};

void fun(int i

void baz(char* str) {
   MyStruct* s = new MyStruct;
   s->fun_ptr = &fun;

   char* buffer = new char[16];
   strcpy(buffer, str);

   s->fun_ptr(5);
```

fun_ptr now points to nopsled/shellcode

Buffer overflow!



Heap metadata

NUL var_a

\xB0 \xCD \xFF \xBF

fun_ptr

A A A A

Heap metadata

A A A A

A A A A

buffer

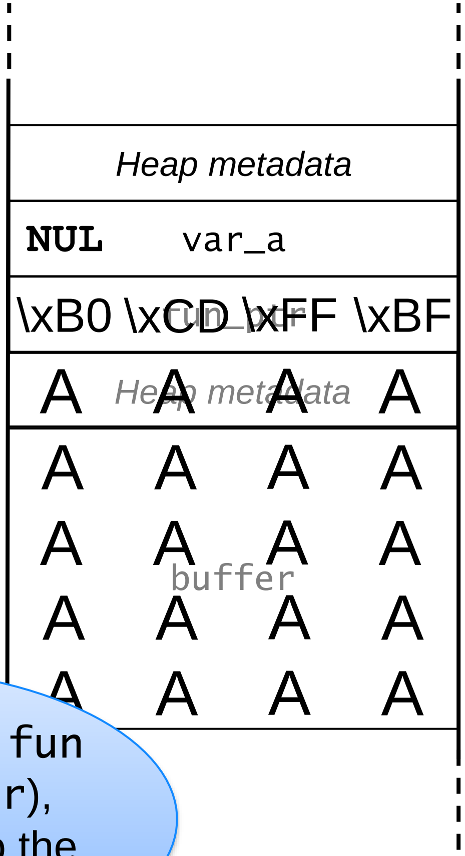
A A A A

A A A A

Function pointer overwrites

Example (C++)

```
struct MyStruct {
   int var_a;
   void (*fun_ptr)(int);
};

void fun(int i);

void baz(char* str) {
   MyStruct* s = new MyStruct;
   s->fun_ptr = &fun;

   char* buffer = new char[16];
   strcpy(buffer, str);

   s->fun_ptr(5);
```



When trying to call fun (through fun_ptr), we instead jump to the shellcode

Function pointer overwrites

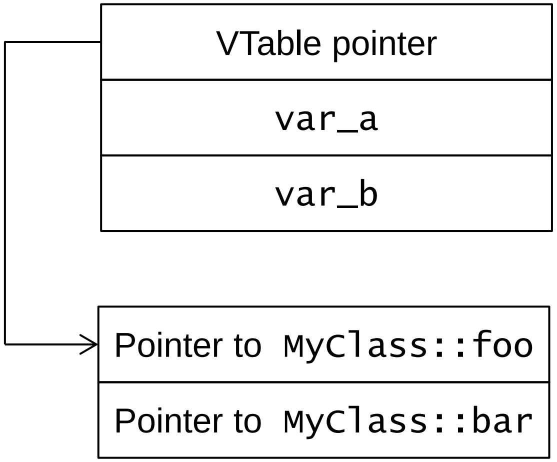
Overwriting C++ VTable pointers

- Objects of classes with virtual functions have an implicit VTable-pointer data member
- The VTable pointer points to a table of function pointers for the specific class.
- Calls to virtual functions are made by looking up corresponding function pointer in VTable during runtime
  - ⇨ Specific class type of object doesn't need be statically known during compilation

- Possible to overwrite VTable pointer to point to a fake VTable using a buffer overflow
  - Not as easy as it may seem!
  - Need to overwrite with a *pointer to a pointer* to desired address
  - May still be possible with various "tricks"

```
class MyClass {
   int var_a;
   int var_b;
   virtual void foo();
   virtual void bar();
};
```

Representation of a MyClass object in memory



VTable pointer

`var_a`

`var_b`

Pointer to `MyClass::foo`

Pointer to `MyClass::bar`

Other heap-related vulnerabilities

Use-after-free

- Program use stale pointer to heap-allocated memory that has already been freed.
- May lead to information disclosure…
  - Attacker can trick program into printing data in freed memory, after it has been re-allocated to store sensitive data
- …or arbitrary code execution
  - Attacker can have program re-allocate freed memory to store attacker-supplied data.
  - If program later use a function pointer or C++ VTable entry in freed object, execution can be redirected by attacker.

Double-free

- Program calls ‘free’ or ‘delete’ on pointer to already freed memory
- Can corrupt memory manager metadata to allow arbitrary code execution

Attacks often requires attacker to set up heap to look in a specific way for exploit to succeed

- “Heap feng shui”

Use-after-free

Example (C++)

```
struct MyStruct {
   int var_a;
   void (*fun_ptr)(int);
};

void fun(int i);

void baz(char* str) {
   MyStruct* s = new MyStruct;
   s->fun_ptr = &fun;
   ...

   delete s;
   char* buffer = new char[16];
   strcpy(buffer, str);
   ...

   s->fun_ptr(5);
```

Function pointer

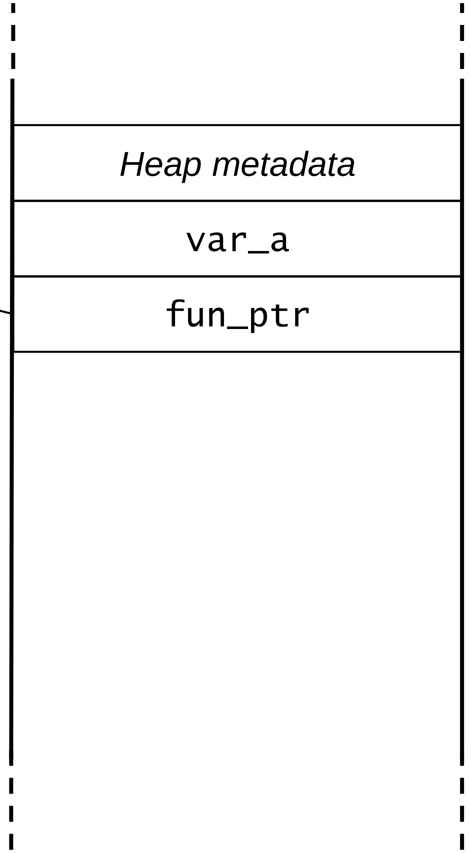
Still heap, not stack

Use-after-free

Example (C++)

```
struct MyStruct {
   int var_a;
   void (*fun_ptr)(in
};

void fun(int i);

void baz(char* str) {
   MyStruct* s = new MyStruct;
   s->fun_ptr = &fun;
   ...

   delete s;
   char* buffer = new char[16];
   strcpy(buffer, str);
   ...

   s->fun_ptr(5);
```

Allocate new `MyStruct` and init `fun_ptr`

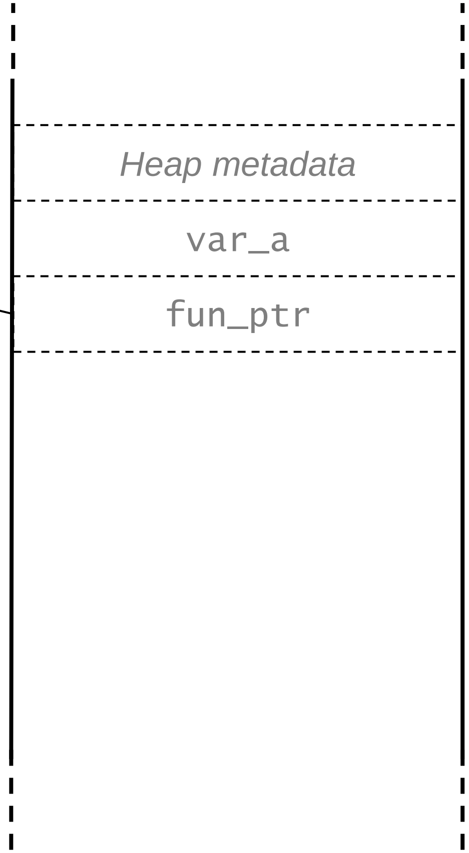


Use-after-free

Example (C++)

```
struct MyStruct {
   int var_a;
   void (*fun_ptr)(int);
};

void fun(int i);

void baz(char
   MyStruct*
   s->fun_ptr =
   ...

   delete s;
   char* buffer = new char[16];
   strcpy(buffer, str);
   ...

   s->fun_ptr(5);
```

Free pointer to the MyStruct

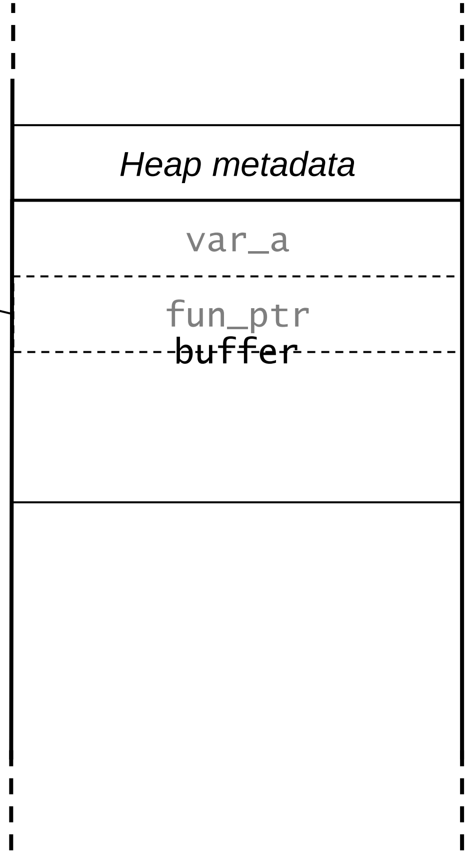


Use-after-free

Example (C++)

```
struct MyStruct {
   int var_a;
   void (*fun_ptr)(int);
};

void fun(int i);

void baz(char*
   MyStruct* s
   s->fun_ptr =
   ...

   delete s;
   char* buffer = new char[16];
   strcpy(buffer, str);
   ...

   s->fun_ptr(5);
```

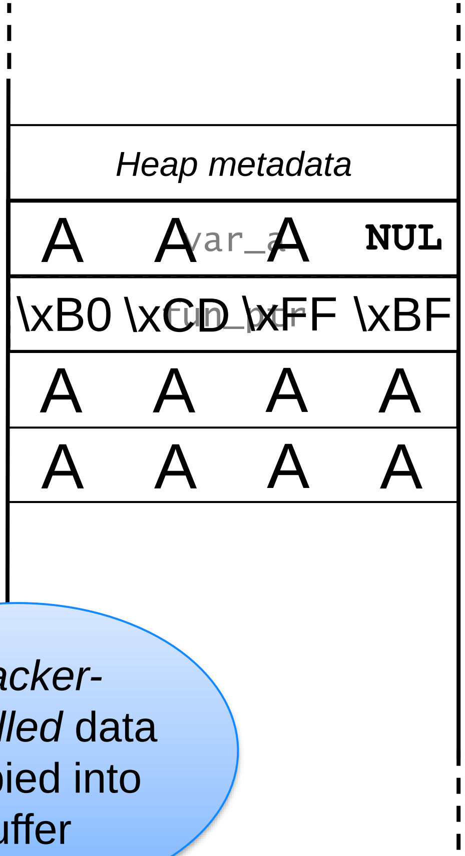
Allocate buffer. Previously freed space is now ***re-used***



Use-after-free

Example (C++)

```
struct MyStruct {
   int var_a;
   void (*fun_ptr)(int);
};

void fun(int i);

void baz(char* str) {
   MyStruct* s = new MyStruct;
   s->fun_ptr = &fun;
   ...

   delete s;
   char* buffer = new char[16];
   strcpy(buffer, str);
   ...

   s->fun_ptr(5);
```



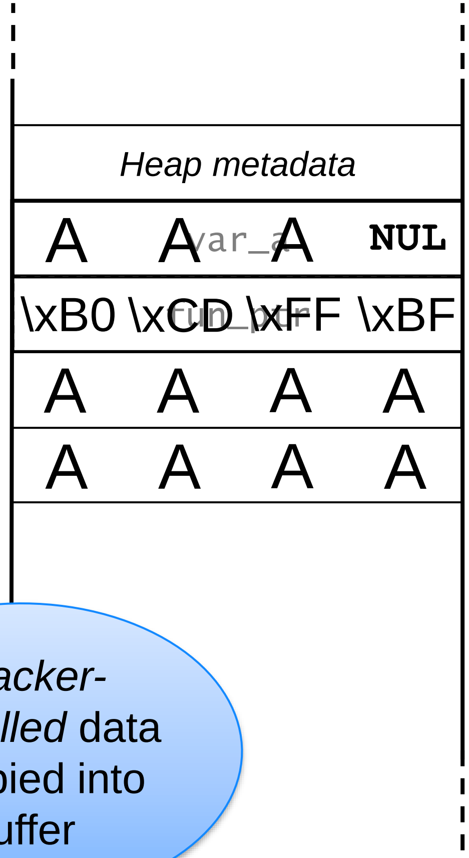
Attacker-controlled data is copied into buffer

Use-after-free

Example (C++)

```
struct MyStruct {
   int var_a;
   void (*fun_ptr)(int);
};

void fun(int i);

void baz(char* str) {
   MyStruct* s = new MyStruct;
   s->fun_ptr = &fun;
   ...

   delete s;
   char* buffer = new char[16];
   strcpy(buffer, str);
   ...

   s->fun_ptr(5);
```

Address to shellcode…



Attacker-controlled data is copied into buffer

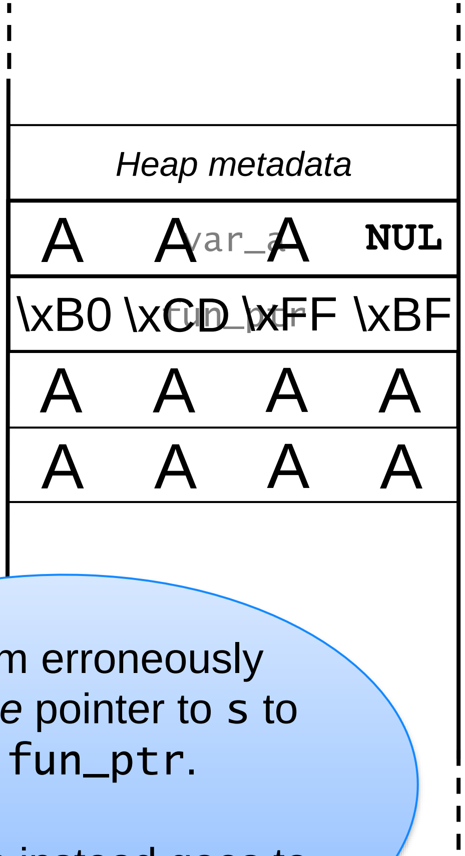
Use-after-free

Example (C++)

```
struct MyStruct {
   int var_a;
   void (*fun_ptr)(int);
};

void fun(int i);

void baz(char* str) {
   MyStruct* s = new MyStruct;
   s->fun_ptr = &fun;
   ...

   delete s;
   char* buffer = new char[16];
   strcpy(buffer, str);
   ...

   s->fun_ptr(5);
```

Program erroneously uses *stale* pointer to s to call `fun_ptr`.

Execution instead goes to shellcode!



Heap metadata

A A A NUL

\xB0 \xCD \xFF \xBF

A A A A

A A A A

Avoiding use-after-free and double-free bugs

- Set pointers to NULL directly after calling free/delete on them to avoid trivial errors.
- In practice, bugs are often caused by pointer aliasing – several pointers pointing to the same memory
    - One component calls free/delete on a pointer, while a different component keeps using another copy of the pointer
    - Avoid passing around pointers to heap-allocated data between different modules.
    - Using the C++ “Resource Allocation Is Initialization” (RAII) pattern avoids confusion about which component is responsible for deallocating data.
    - Use “smart pointers” with reference counting where applicable, (e.g. with respect to performance)