Buffer Overflow Attacks

Computer Systems 3.10.3-4,

Hacking

- Roots in phone phreaking
- White Hat vs Gray Hat vs Black Hat
- Over 50% of Modern Software Development is Black Hat!

Tip the balance: Be a force for good… not evil!

Disclaimer – Buffer Overflow Attack

- DO NOT ABUSE!
- Modern code is protected from this attack several ways
- Ancient form of hacking
  - First documented in 1972
  - Used in 1988 “Morris Worm” – First internet virus
  - Used to hack Unix, Windows, Xbox, PS2, Wii
- Taught here as an example of what to watch out for!

Example Vulnerable Code

```
bool getString() {
   char buffer[81];
   buffer[0]='\0';
   gets(buffer);
   if (strlen(buffer)>0) {
      printf("Read line: %s\n",buffer);
      return true;
   }
   return false;
}
```

- "gets" reads from stdin until it finds either an end-of-file or a newline (which is replaced by a null terminator).
- "gets" copies whatever it reads into the argument (buffer).
- "gets" does not check to make sure result fits in space allocated.

x86 expansion

```
bool getString() {
   char buffer[81];
   buffer[0]='\0';
   gets(buffer);
   if (strlen(buffer)>0) {
      printf("Read line: %s\n",buffer);
      return true;
   }
   return false;
}
```

```
push   %rbp
mov    %rsp,%rbp
sub    $0x60,%rsp
```

```
movb   $0x0,-0x60(%rbp)
```

```
lea    -0x60(%rbp),%rax
mov    %rax,%rdi
callq  4004c0 <gets@plt>
```

```
lea    -0x60(%rbp),%rax
movzbl (%rax),%eax
test   %al,%al
je     400649 <getString+0x40>
```

```
mov    $0x1,%eax
jmp    40064e <getString+0x45>
```

```
lea    -0x60(%rbp),%rax
mov    %rax,%rsi
mov    $0x40071f,%edi
mov    $0x0,%eax
callq  400490 <printf@plt>
```

```
mov    $0x0,%eax
leaveq
retq
```

stack after call to getString

```
...
        while( (gs=getString())) {};
833:    b8 00 00 00 00      mov    $0x0,%eax
838:    e8 1c 00 00 00      callq  859 <getString>
83d:    88 45 ff            mov    %al,-0x1(%rbp)
840:    80 7d ff 00         cmpb   $0x0,-0x1(%rbp)
844:    75 ed               jne    833 <main+0x23>
...
```

main's stack frame

return address

%rbp

%rsp

Address	Value (64 bit)
0000 7FFF FFFF E890	0000 7FFF FFFF E980
0000 7FFF FFFF E888	0000 0001 5555 46E0
0000 7FFF FFFF E880	0000 7FFF FFFF E988
0000 7FFF FFFF E878	0000 5555 5555 483D
0000 7FFF FFFF E870	0000 0000 0000 0000
0000 7FFF FFFF E868	0000 0000 0000 0000
...	...
0000 7FFF FFFF E828	0000 0000 0000 0000
0000 7FFF FFFF E820	0000 0000 0000 0000
0000 7FFF FFFF E818	0000 7FFF F7DD 4600
0000 7FFF FFFF E810	0000 0000 0000 0000
0000 7FFF FFFF E808	0000 7FFF F7AA EDB2
0000 7FFF FFFF E800	0000 0000 0000 0000
....	

stack in getString after preamble

```
...
        bool getString() {
859:    55                  push   %rbp
85a:    48 89 e5            mov    %rsp,%rbp
85d:    48 83 ec 60         sub    $0x60,%rsp
        char buffer[81];
        buffer[0]='\0';
861:    c6 45 a0 00         movb   $0x0,-0x60(%rbp)...
```

main's stack frame

return address

%rbp

main's %rbp

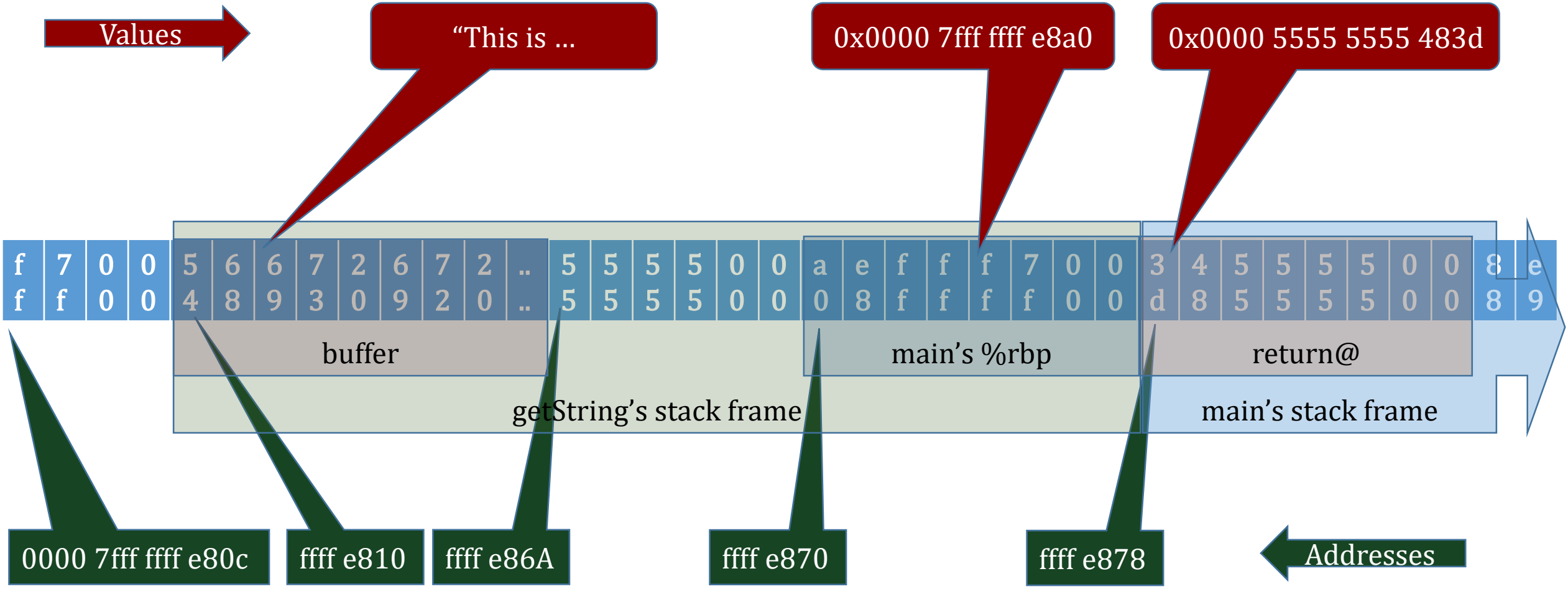
getString's stack frame

buffer

%rsp

Address	Value (64 bit)
0000 7FFF FFFF E890	0000 7FFF FFFF E980
0000 7FFF FFFF E888	0000 0001 5555 46E0
0000 7FFF FFFF E880	0000 7FFF FFFF E988
0000 7FFF FFFF E878	0000 5555 5555 483D
0000 7FFF FFFF E870	0000 7FFF FFFF E8A0
0000 7FFF FFFF E868	0000 0000 0000 0000
...	...
0000 7FFF FFFF E828	0000 0000 0000 0000
0000 7FFF FFFF E820	0000 0000 0000 0000
0000 7FFF FFFF E818	0000 7FFF F7DD 4600
0000 7FFF FFFF E810	0000 0000 0000 0000
0000 7FFF FFFF E808	0000 7FFF F7AA EDB2
0000 7FFF FFFF E800	0000 0000 0000 0000
....	

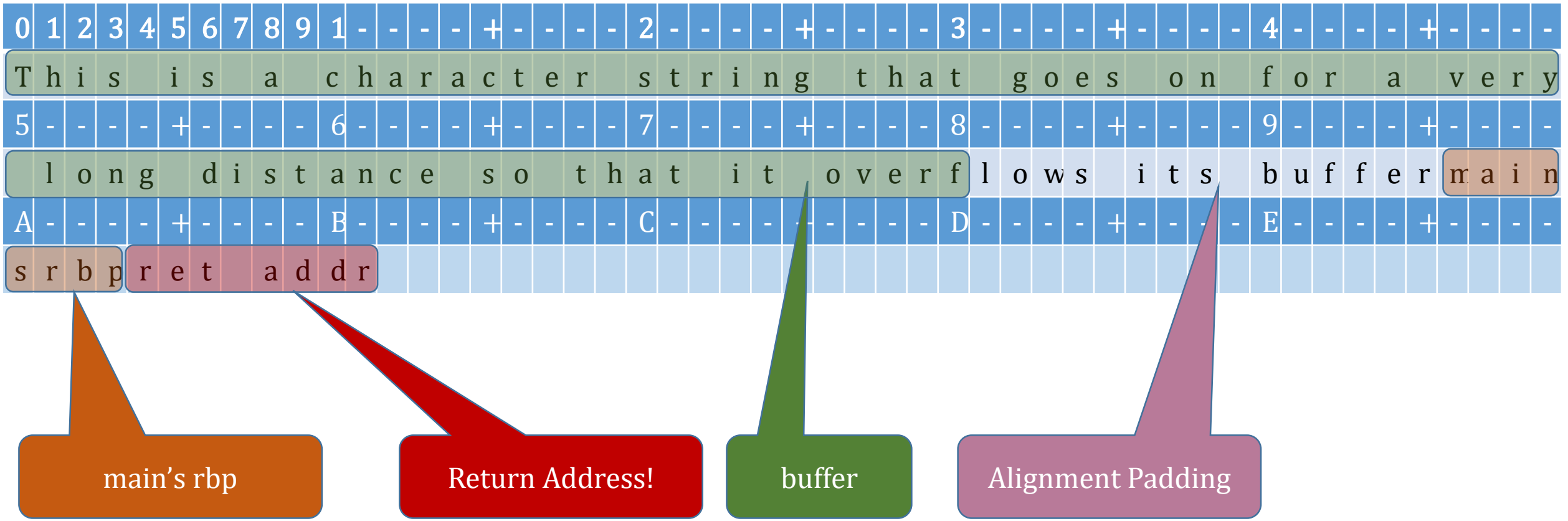
Stack left to right (little endian)



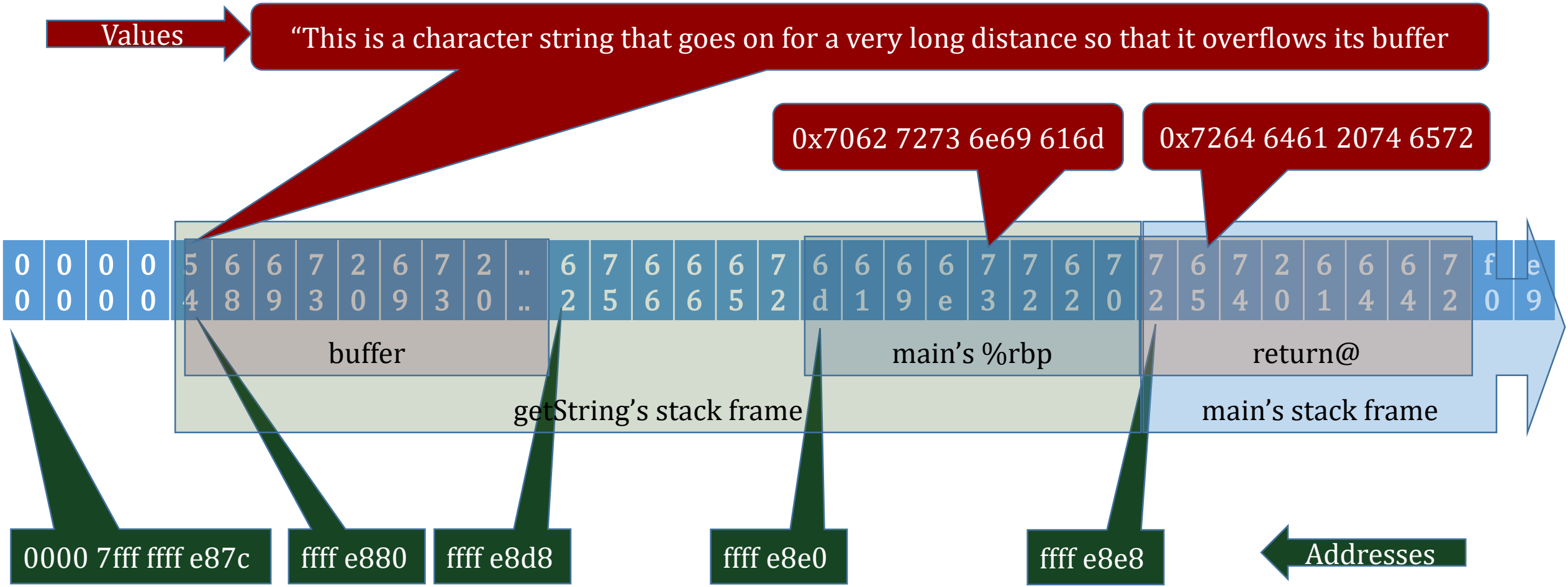
Buffer Overflow

- Put a very long string into an input file
- Much longer than the length of “buffer”
- So long that it writes over what is past “buffer” in getString’s frame
  - What is past “buffer”?
- So long that it writes over the lowest address in main’s frame!
  - What is in the lowest address in main’s frame?

“String” in file (stdin) read by gets



Stack left to right (little endian)



stack in getString after gets

```
...
        gets(buffer);
865:    48 8d 45 a0         lea   -0x60(%rbp),%rax
869:    48 89 c7            mov    %rax,%rdi
86c:    e8 3f fe ff ff      callq  6b0 <gets@plt>
        if (strlen(buffer)>0) {
871:    48 8d 45 a0         lea   -0x60(%r[illegible]),%[illegible]
...
```

main's stack frame

return address

%rbp

main's %rbp

getString's stack frame

buffer

%rsp

Address	Value (64 bit)
0000 7FFF FFFF E890	0000 7FFF FFFF E980
0000 7FFF FFFF E888	0000 0001 5555 46E0
0000 7FFF FFFF E880	0000 7FFF FFFF E988
0000 7FFF FFFF E878	7264 6461 2074 6572
0000 7FFF FFFF E870	7062 7273 6e69 616d
0000 7FFF FFFF E868	7265 6666 7562 6220
...	...
0000 7FFF FFFF E828	2074 6168 7420 676e
0000 7FFF FFFF E820	6972 7473 2072 6574
0000 7FFF FFFF E818	6361 7261 6863 2061
0000 7FFF FFFF E810	2073 6920 7369 6854
0000 7FFF FFFF E808	0000 7FFF F7AA EDB2
0000 7FFF FFFF E800	0000 0000 0000 0000
....	

stack in getString at return

```
...
                    return true;
 894:    b8 01 00 00 00      mov   $0x1,%eax
 899:    eb 05               jmp   8a0 <getString+0x47>
...
8a0:    c9                  leaveq
 8a1:    c3                  retq ...
```

return address

%rsp

%rip (value is 0x7264 6461 2074 6572)

Segmentation Fault

Address	Value (64 bit)
0000 7FFF FFFF E890	0000 7FFF FFFF E980
0000 7FFF FFFF E888	0000 0001 5555 46E0
0000 7FFF FFFF E880	0000 7FFF FFFF E988
0000 7FFF FFFF E878	7264 6461 2074 6572
0000 7FFF FFFF E870	7062 7273 6e69 616d
0000 7FFF FFFF E868	7265 6666 7562 6220
...	...
0000 7FFF FFFF E828	2074 6168 7420 676e
0000 7FFF FFFF E820	6972 7473 2072 6574
0000 7FFF FFFF E818	6361 7261 6863 2061
0000 7FFF FFFF E810	2073 6920 7369 6854
0000 7FFF FFFF E808	0000 7FFF F7AA EDB2
0000 7FFF FFFF E800	0000 0000 0000 0000
....	

Mixing Hex and ASCII

- Normally treat a file as a string of ASCII characters
- In fact, each ASCII character has a hex representation…

T	h	i	s		i	s		a		c	h	a	r	a	c	…
54	68	69	73	20	69	73	20	61	20	63	68	61	72	61	63	…

- We can write a program to put non-ASCII hex data in a file
  - See xmp_mix
- Use the command "od –Ax –t x1z" to show both ASCII and hex

```
000000 54 68 69 73 20 69 73 20 61 20 63 68 61 72 61 63  >This is a charac<
000010 74 65 72 20 73 74 72 69 6e 67 de ad be ef 0a     >ter string.....<
00001f
```

Example of a file with ASCII and Hex

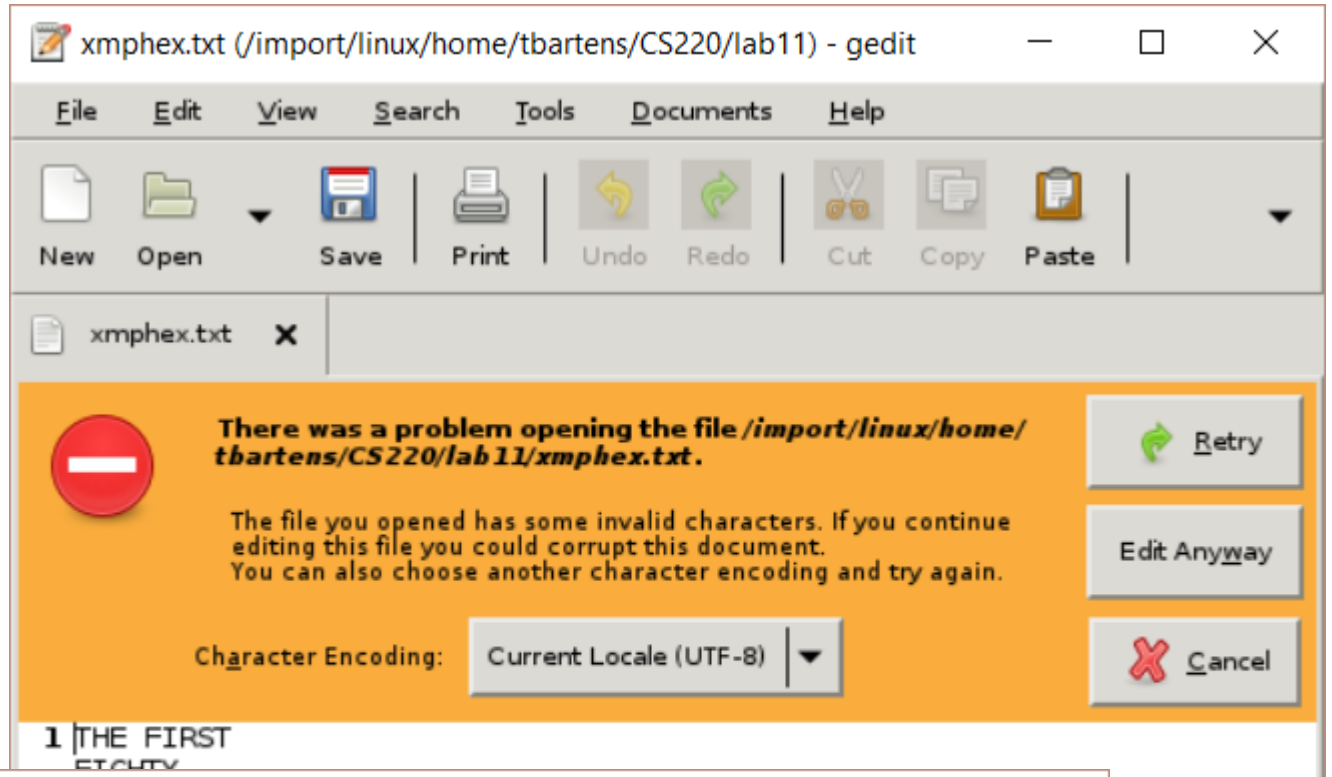
- ASCII Representation on terminal “cat file”…

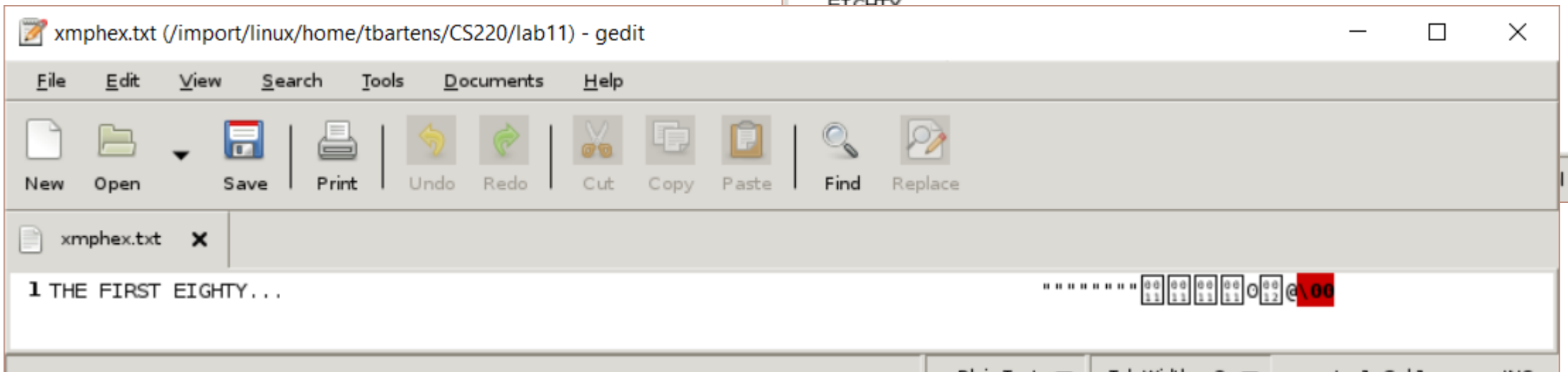
```
This is a character stringÞ¾ï
```

- Mixed representation “od –Ax -t x1z file”

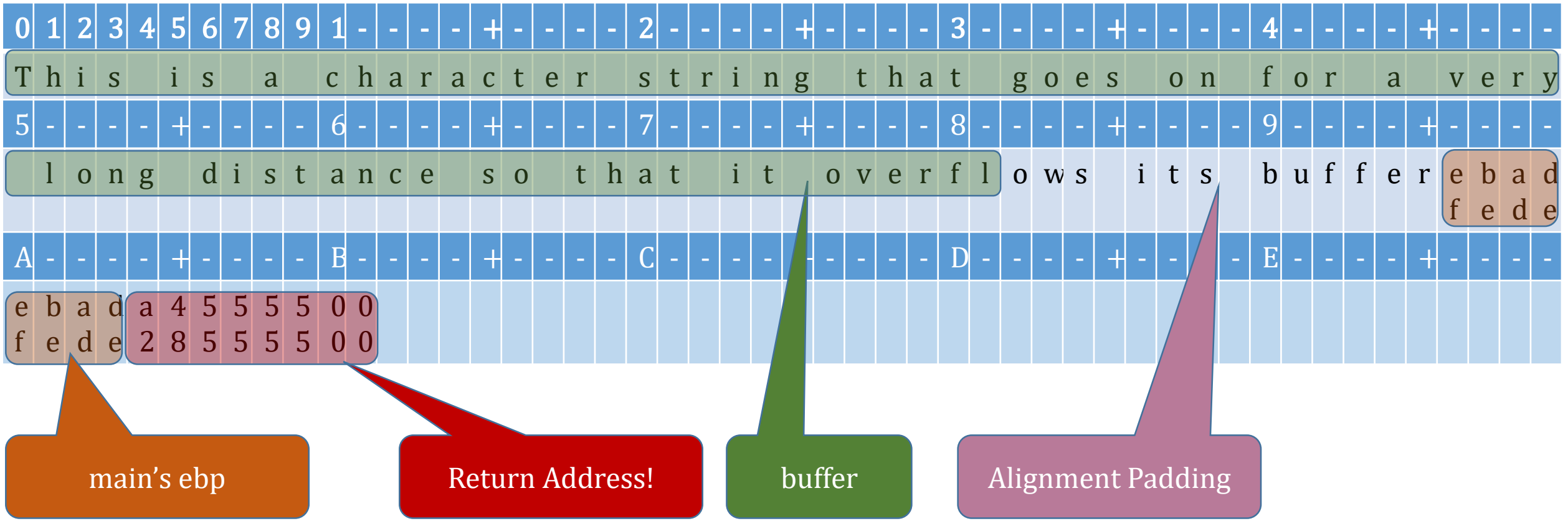
```
000000 54 68 69 73 20 69 73 20 61 20 63 68 61 72 61 63  >This is a charac<
000010 74 65 72 20 73 74 72 69 6e 67 de ad be ef 0a     >ter string.....<
00001f
```

GEDIT Mixed File

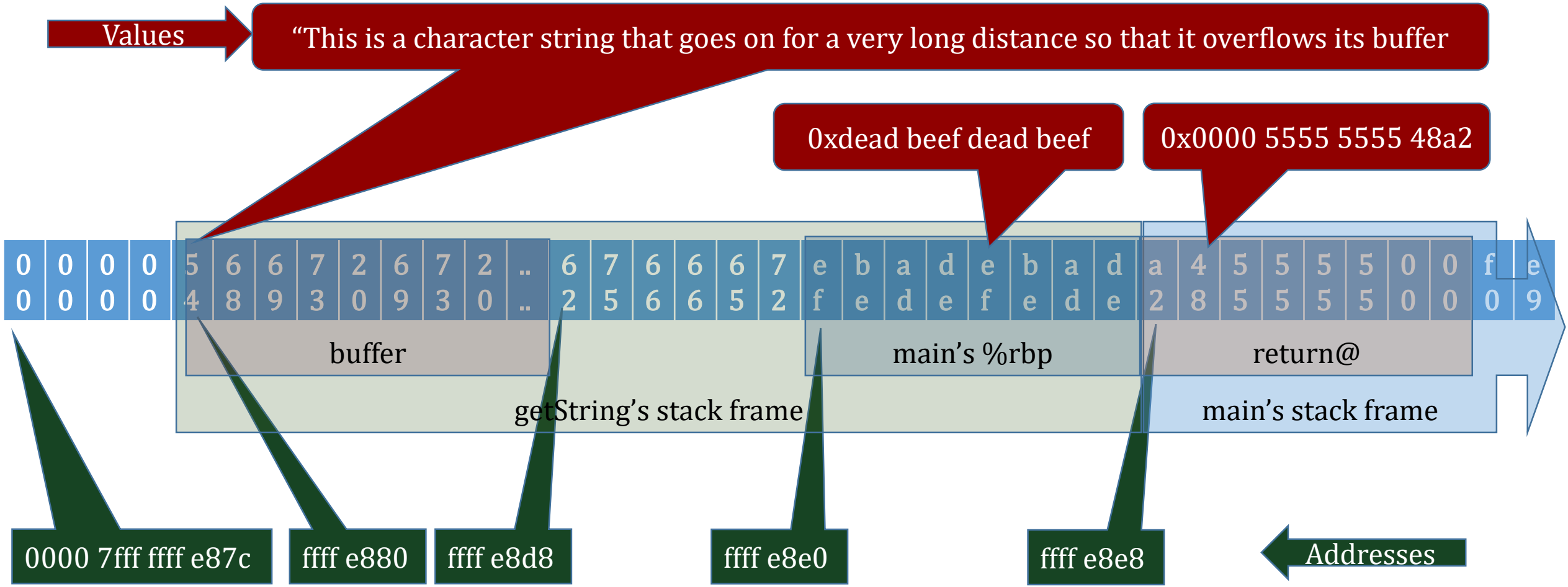




“String” in file (stdin) read by gets



Stack left to right (little endian)



stack in getString after gets

```
...
        gets(buffer);
865:    48 8d 45 a0         lea   -0x60(%rbp),%rax
869:    48 89 c7            mov    %rax,%rdi
86c:    e8 3f fe ff ff      callq  6b0 <gets@plt>
        if (strlen(buffer)>0) {
871:    48 8d 45 a0         lea   -0x60(%r
...
```

main's stack frame

return address

%rbp

main's %rbp

getString's stack frame

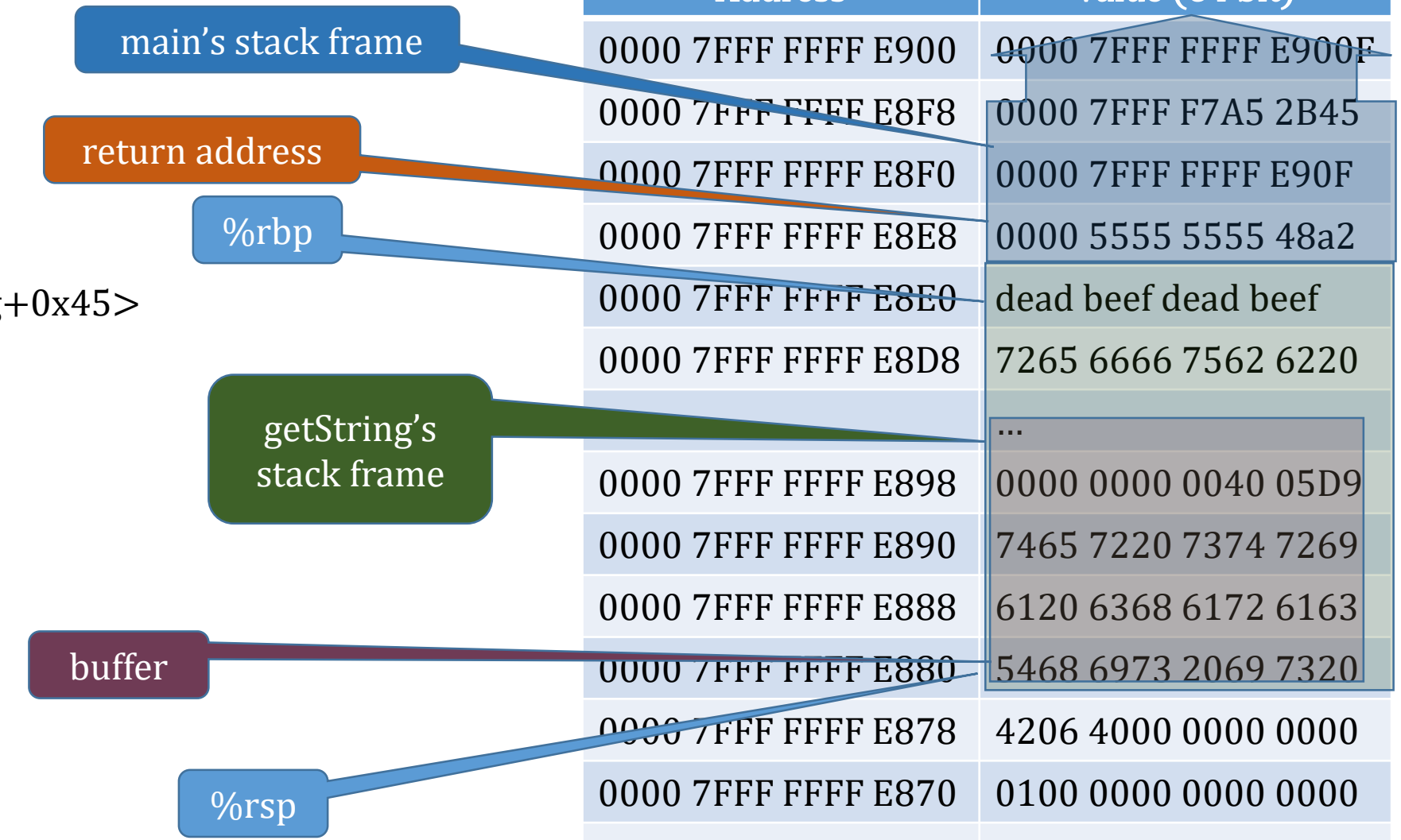
buffer

%rsp

Address	Value (64 bit)
0000 7FFF FFFF E890	0000 7FFF FFFF E980
0000 7FFF FFFF E888	0000 0001 5555 46E0
0000 7FFF FFFF E880	0000 7FFF FFFF E988
0000 7FFF FFFF E878	0000 5555 5555 48a2
0000 7FFF FFFF E870	dead beef dead beef
0000 7FFF FFFF E868	7265 6666 7562 6220
...	...
0000 7FFF FFFF E828	2074 6168 7420 676e
0000 7FFF FFFF E820	6972 7473 2072 6574
0000 7FFF FFFF E818	6361 7261 6863 2061
0000 7FFF FFFF E810	2073 6920 7369 6854
0000 7FFF FFFF E808	0000 7FFF F7AA EDB2
0000 7FFF FFFF E800	0000 0000 0000 0000
....	

stack in getString after gets

```
...
mov    $0x1,%eax
jmp    40064e <getString+0x45>
...
leaveq
retq
```



Address	Value (64 bit)
0000 7FFF FFFF E900	0000 7FFF FFFF E90F
0000 7FFF FFFF E8F8	0000 7FFF F7A5 2B45
0000 7FFF FFFF E8F0	0000 7FFF FFFF E90F
0000 7FFF FFFF E8E8	0000 5555 5555 48a2
0000 7FFF FFFF E8E0	dead beef dead beef
0000 7FFF FFFF E8D8	7265 6666 7562 6220
	...
0000 7FFF FFFF E898	0000 0000 0040 05D9
0000 7FFF FFFF E890	7465 7220 7374 7269
0000 7FFF FFFF E888	6120 6368 6172 6163
0000 7FFF FFFF E880	5468 6973 2069 7320
0000 7FFF FFFF E878	4206 4000 0000 0000
0000 7FFF FFFF E870	0100 0000 0000 0000
....	

What could be at return address?

Your evil code!

Problems with Buffer Overflow Attack

- Need to know offset from buffer to top of stack / return @
- Need to mix ASCII with hexadecimal in input file
- String can't contain "newline" (0x0A)
- Need to know address of pirate routine to "return" to
- Hard to "return" to conventional flow after attack… return @ overwritten
- %rbp has been overwritten – lost top of caller's stack frame
- Randomized load uses offsets, not absolute addresses

- Get that from objdump –d
- Can write a program to do that
- Basic restriction
- Challenge for project 4
- Can get original return@ from objdump –d, or just don't return
- %rsp points to bottom of caller's frame, and we can find its size from objdump –d, or just don't use
- Turn off randomized load

Address Space Layout Randomization

- ASLR is a feature introduced to prevent things like buffer overflow attacks
- Randomizes where your code is loaded
  - Every time you execute, your function is loaded at a different address
  - Hence objdump no longer prints complete address – just last three digits
- ASLR will prevent "return" to a different function in the same code
  - function instructions are at a different place each run
- gdb turns off ASLR
  - code is always loaded at 0x0000 5555 5555 xxxx
- To turn off ASLR on LDAP: >setarch linux64 –R <command>

Preventing Buffer Overflow w/ "Guard"

```
bool getString() {
    struct bufStr {
        char buffer[81];
        int guard;
    } buf = { { 0x00 } { 0xFEDCBA98 }} ;
    gets(buf.buffer);
    if (strlen(buffer)>0) {
        assert(buf.guard==0xFEDCBA98);
        printf("Read line: %s\n",buf.buffer);
        return true;
    }
    return false;
}
```

- guard goes in stack frame after (on top of) buffer
- If buffer overflow occurs, guard will be modified
- If buffer overflow occurs, assert will fail
- Unless hacker did objdump and put the guard value in his hacked file

gcc stack guarding

- If you compile with the –fstack-protector flag, gcc will automatically provide stack guards for you

- If gcc finds a corrupted stack guard, it prints…

```
*** stack smashing detected ***: ./target terminated
Segmentation fault
```

- gcc stack protection is OFF by default on LDAP machines

Preventing Buffer Overflow w/ fgets

```
bool getString() {
   char buffer[81];
   buffer[0]='\0';
   fgets(buffer,sizeof(buffer),stdin);
   if (strlen(buffer)>0) {
      printf("Read line: %s\n",buffer);
      return true;
   }
   return false;
}
```

- “fgets” reads from any stream (third parameter)
- “fgets” reads until EITHER newline (0x10), OR second parameter reached.
- “fgets” prevents buffer overwrite

Preventing w/ Memory Protection

- Your entire address space is divided into 4096 byte "pages"
  - $4096=2^{12}$; It takes 12 bits, or 3 hex digits to count from 0 to 4095
  - 64 bit addresses… first 52 bits (13 hex characters) are page number
  - 64 bit addresses… last 12 bits (3 hex characters) are offset within page
- Each page in memory has three independent security attributes:
  - Can I read memory in this page
  - Can I write memory in this page
  - Can I execute instructions in this page
- If you try to perform an action on a page which is not allowed, you will get a segmentation violation

Default Memory Protection

- When you code is loaded into memory, it is loaded in pages that allow read and execute, but not write
- Pages in the stack allow read or write, but not execution
- Pages on the heap (malloc'ed space) allow read or write, but not execution
- There are library routines to modify memory protection for a page, but by default, there is no memory you are allowed to both write to, and execute from.
- If your evil code is not already loaded, it's hard to execute it!
  - Project 4 has special code to avoid this problem!