Chapter 6. Format String Bug

Prof. Jaeseung Choi
Dept. of Computer Science and Engineering
Sogang University

Format String Bug

- **Another classic type of vulnerability**
  - Almost as old as buffer overflow
- **Caused by misuse of `printf()`-like functions that take in format specifier and variable arguments**
- **It is not widespread anymore, but it gives us several meaningful lessons**

Our Old Friend `printf()`

- **You might have used it even in your first C program**
  - Convenient for printing our various types
- **One unique feature of `printf()` is that it can take in variable number of arguments**
  - Number of arguments must agree with the number of format specifiers (%d, %c, %s ...) in the first argument

```
int main(void) {
    int i = 10;
    char c = 'A';
    printf("Hello world\n");
    printf("i = %d, c = %c\n", i, c);
    return 0;
}
```

Internals of `printf()`

- **The prototype of `printf()` is declared as follow**

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

- **The first argument `char *format` is called format string**
- **`printf()` processes this format string and consumes additional arguments one by one**
  - Every time a **format specifier** (%d, %c, %s ...) is encountered, convert the next argument into a string and print it

```
int printf(const char *format, ...) {
    do {
        // Process format string and args
    } while (?)
}
```

Common Mistake

- **What happens if the number format specifiers do not match with the number of provided values?**
  - Three format strings **`%d`**, **`%c`**, **`%x`** vs. two values **`i`**, **`c`**
- **Although the compiler may print out some warnings, the program below will compile and run**
  - What will be printed as the third value?
  - `printf()` will think that there is additional argument for **`%x`**

```c
int main(void) {
    int i = 10;
    char c = 'A';
    printf("%d %c %x\n", i, c);
    return 0;
}
```

Common Mistake: At Low-level

- **In x86-64 Linux system, `printf()` will fetch the value in register `%rcx`**
  - (Review) In x86-64 calling convention, the first 6 arguments are passed through **`%rdi, %rsi, %rdx, %rcx, %r8, %r9`**. And the next arguments will be passed through the stack
- **As a result, the value of this register will be printed out**
  - This value must have been initialized before **`main()`** is called

```c
int main(void) {
    int i = 10;
    char c = 'A';
    printf("%d %c %x\n", i, c);
    return 0;
}
```

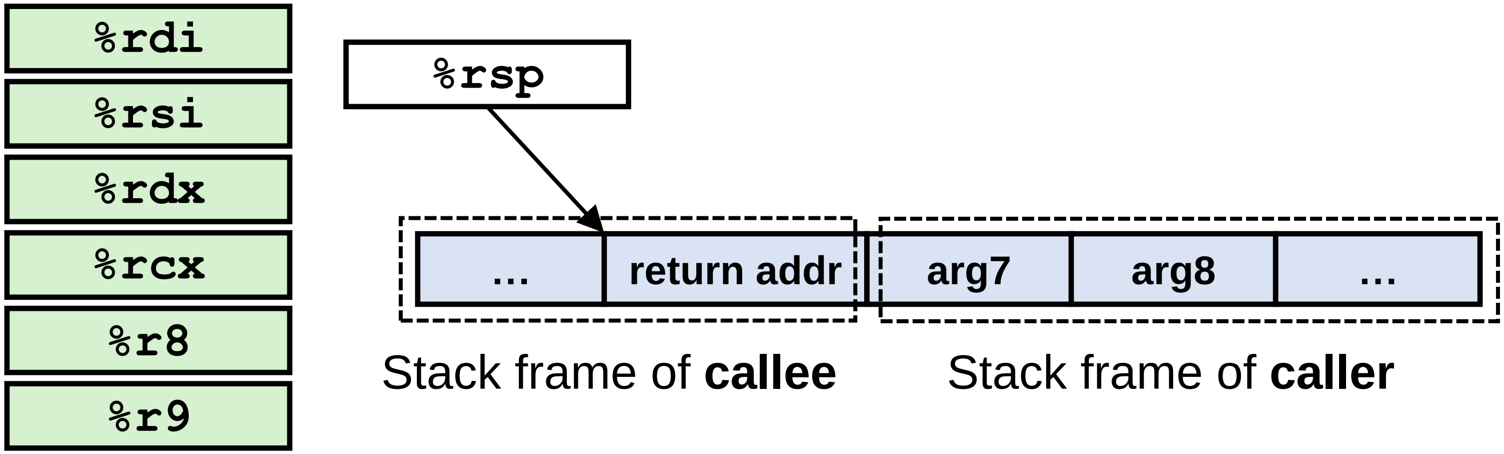
More Serious Mistake

- **Let's assume a simple program that uses `fgets()` to prevent buffer overflow vulnerability**
- **But this time, the programmer was too lazy to type in the whole `printf("%s", buf);` part**
- **How about writing the code more concisely like below?**
  - This is called **format string bug**, and hackers can exploit this!

```
int main(void) {
    char buf[64];
    fgets(buf, sizeof(buf), stdin);
    // printf("%s", buf);
    printf(buf); // Format string bug
    return 0;
}
```

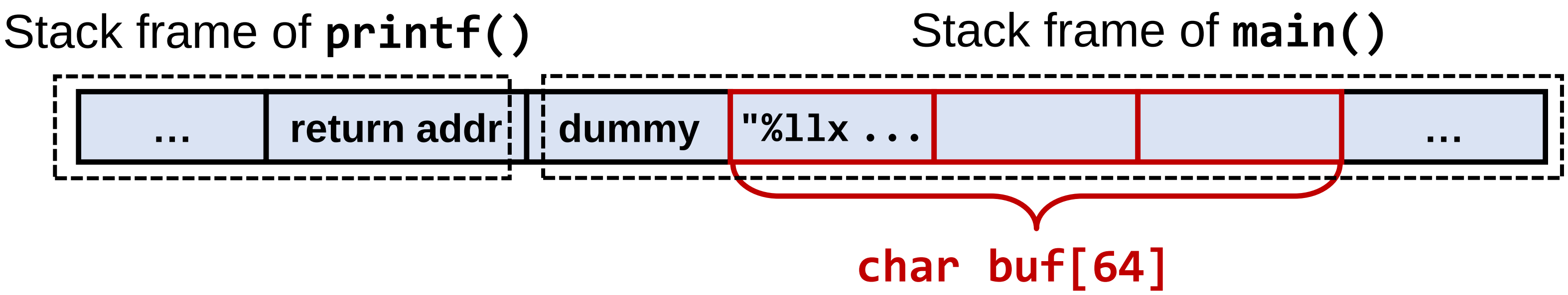
Format String Bug (FSB)

- **By entering `%llx` for 5 times, we can dump the values of register from `%rsi` to `%r9`**
  - Can use any specifier; just chose **`%llx`** to print the whole 8-byte
- **What if we continue to enter `%llx` in the format string?**
  - 7th, 8th, … arguments will be fetched from the stack
  - Of course, such arguments are actually **not** provided
  - So it will disclose the content of stack instead



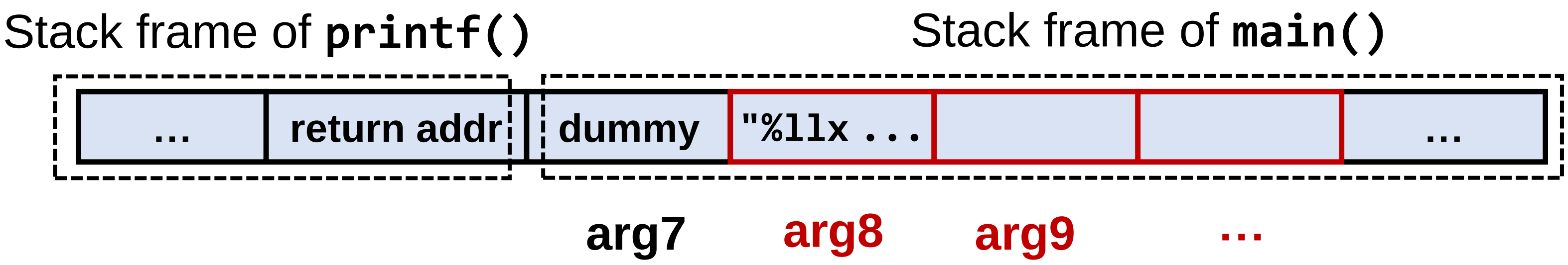
FSB: Disclosing Other Areas

- **Then, can we only disclose the stack area?**
- **Often, you can also dump *arbitrary* addresses**
- **If we provide even more format specifiers, `printf()` will eventually reach the local buffer and consume it**
  - Let's assume that our example has the following stack frames
  - Note that **`buf[64]`** will contain the string provided by the hacker (e.g., a string that starts with `"%llx %llx ..."`)
  - Due to the limited space, some blocks are omitted here



FSB: Disclosing Other Areas

- **Then, can we only disclose the stack area?**
- **Often, you can also dump *arbitrary* addresses**
- **If we provide even more format specifiers, `printf()` will eventually reach the local buffer and consume it**
  - Then, if the hacker provides many format specifiers, **`printf()`** will interpret the **`buf[64]`** area as arg8, arg9, …
  - What if the hacker initializes one of the argument (e.g., arg14) as **`0x414243`**, and make it consumed by **`%s`** format specifier?
  - Characters stored in address **`0x414243`** will be printed out!



FSB: Overwriting Memory?

- **So hackers can read from arbitrary memory address**
  - But hackers cannot write to arbitrary memory address, right?
- **Unfortunately, overwriting memory is also possible**
  - By using **%n** or **%hn**: you must not have heard of these before
  - These format specifiers let you store the ***number of character bytes printed so far***

```
int main(void) {
    int i, j;
    printf("ABCDE12345%n\n", &i); // i = 10
    printf("%d%n\n", 100, &j); // j = 3
    printf("i = %d, j = %d\n", i, j);
    return 0;
}
```

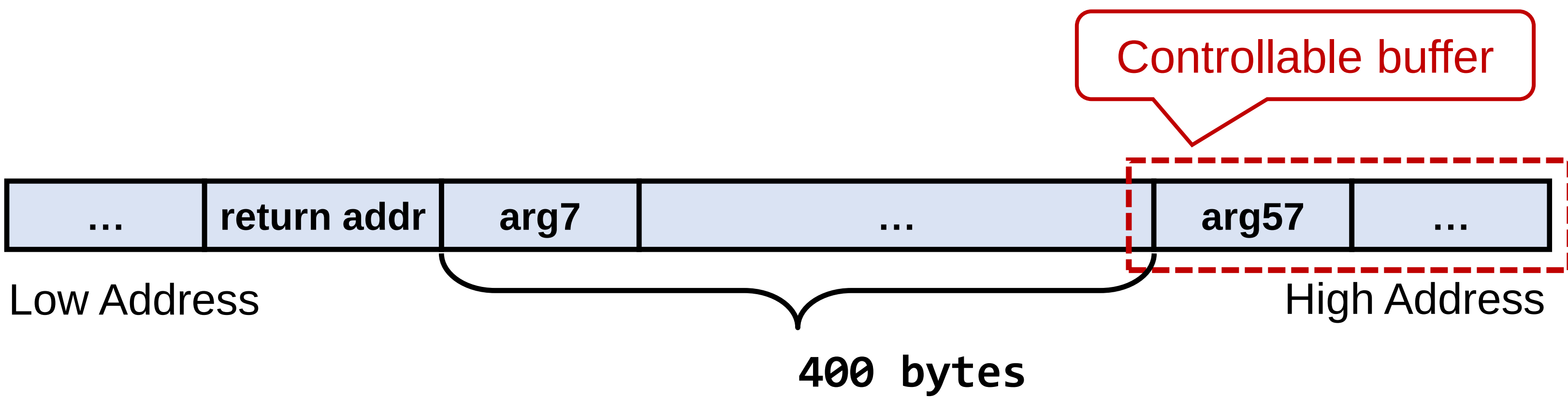
From FSB to Control Hijack

- **Using %n, we can write to arbitrary memory address, as we used %s to read from arbitrary memory address**
- **This allows us to hijack the control-flow of a program**
  - Ex) By overwriting saved return address or GOT entry
- **For this, we must control the value that is written to the address that we chose**
  - We can use **width field** to control the ***number of printed bytes***

```
int main(void) {
    int i;
    printf("%5000d %n\n", 100, &i); // i = 5001
    return 0;
}
```

Another Feature of `printf()`

- **You can directly access *(n+1)*-th argument at once**
  - `printf("%2$d", 100, 200, 300, 400) // prints "200"`
- **Now, assume that we are trying to use `%s` (or `%n`) to read (or write) an arbitrary memory address**
- **What if the controllable buffer is far away in the stack?**
  - In the example below, should we first enter **`%d`** for 56 times?
  - Instead, we can just use **`"%56$s"`** to consume arg57 directly



Wrap-up of FSB Attack Scenario

- **If the attacker can control the format string passed to `printf()`, we can read or write memory**
  - By giving **%d** as input, we can dump values in register and stack
  - By giving **%s**, we can read the memory pointed by such values
    - If the consumed value (imaginary argument) is controllable by us, we can read arbitrary memory address
  - By giving **%n**, we can write to the memory pointed by such values
    - Similarly, we can choose which address to overwrite
    - Also, we can use the **width field** to choose the value to write
  - If the controllable buffer is too far away, we can utilize **$** sign

```c
char buf[64];
fgets(buf, sizeof(buf), stdin);
printf(buf); // Format string bug
```

FSB in Real-world Software

- **In 2012, format string bug was found in `sudo` program***
  - Of course, the developers did not "**`printf(user_buffer)`**"
  - The format string **`fmt2`** passed to `fprintf()` was dynamically constructed, and there was a mistake in this point
    - Although **`fmt`** was safe, **`argv[0]`** was user-controllable
    - But wait, isn't **`argv[0]`** always a fixed string, "**`sudo`**"?
    - Attacker can manipulate it by using *symbolic link*
- **Since `sudo` is has SUID bit, one can spawn a shell with *root* privilege if the control flow is hijacked to `execve()`**

```
...
sprintf(fmt2, "%s: %s", argv[0], fmt);
fprintf(stderr, fmt2, ...);
```

* Assigned CVE-2012-0809

Where did it start to go wrong?

- **C programming language and library was designed in a too generous (permissive) way**
- **Maybe it was not a good idea to allow a non-constant value as a format string argument of `printf()`**
  - Many modern languages only allow constant format strings
- **Even if we allow non-constant format string, there is still a chance to catch an error at runtime**
  - By tracking the number of arguments that are actually passed
  - But this is also not supported in C language ☹

Lessons

- **Design of programming language is important**
  - When the compiler of some language rejects your program, don't hate the compiler too much
- **Adding more features may not always be a good idea**
  - Did you know that features like **%n**, **%hn**, or **$** even existed?
  - These features only provided useful attack vectors to hackers
  - Think twice before you add a new feature to your program
- **And once again, attacker (hackers) are persistent and creative in finding ways to exploit software**