CS 370: Introduction to Security

06.01: Software Security II

Tu/Th 4:00 – 5:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab

Topics for today

- Software security
  - Memory safety vulnerabilities
    - Buffer overflow vuln.
    - Integer overflow vuln.
    - Format string vuln.
    - Heap vuln.
    - Off-by-one vuln.
  - Practices to reduce software vulnerabilities

Integer overflow

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

Format string vulnerabilities

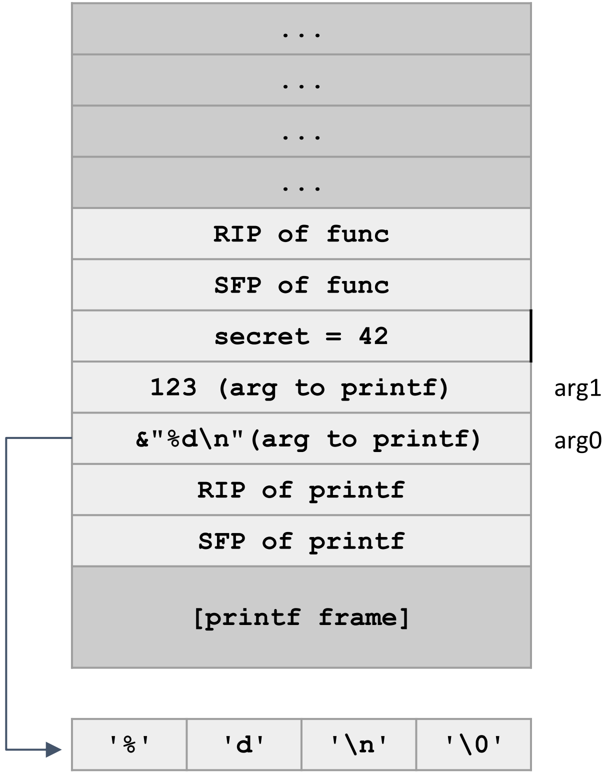
Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

REVIEW: PRINTF FUNCTION

```
void func(void) {
    int secret = 42;
    printf("%d\n", 123);
}
```

printf assumes that there is 1 more argument because there is one format sequence and will look 4 bytes up the stack for the argument

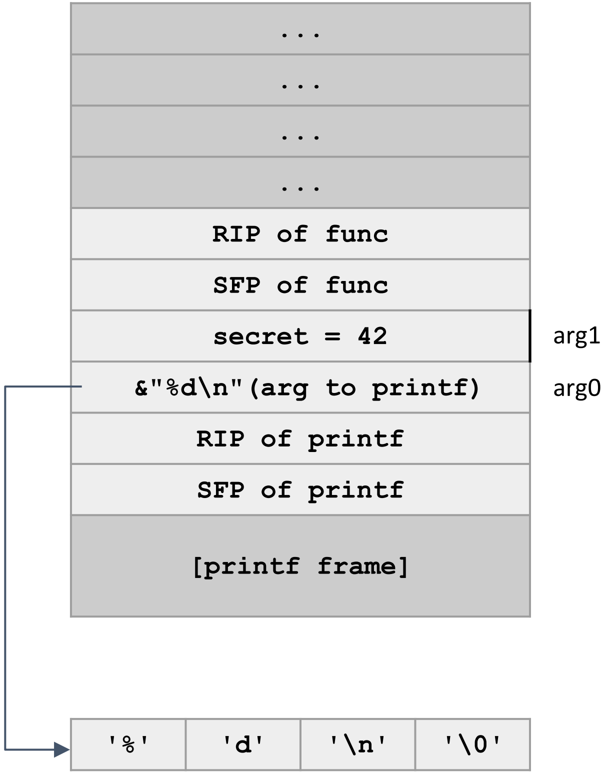


REVIEW: PRINTF FUNCTION

```
void func(void) {
    int secret = 42;
    printf("%d\n", 123);
}
```

`printf` assumes that there is 1 more argument because there is one format sequence and will look 4 bytes up the stack for the argument

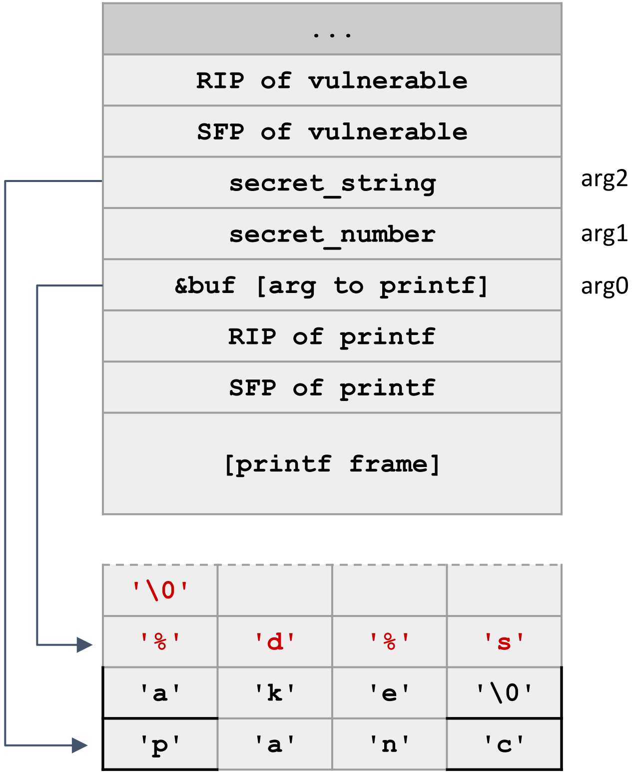
Because the format string contains the **`%d`**, it will still look 4 bytes up and print the value of **`secret`**!



Format string vulnerabilities

```
char buf[64];

void vulnerable(void) {
    char *secret_string = "pancake";
    int secret_number = 42;
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

If we use **`printf("%d%s")`**. **`printf`** reads its first argument (arg0), sees two format specifiers, and expects two more arguments (arg1 and arg2).



Format string vulnerabilities – cont'd

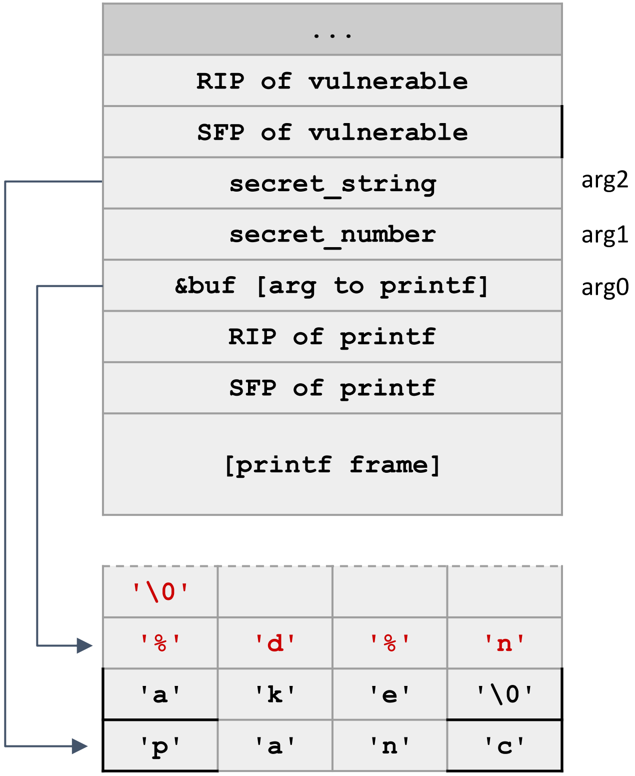
```
char buf[64];

void vulnerable(void) {
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

- The attacker can also write values using the `%n` specifier
  - `%n` treats the next argument as a pointer and writes the # of bytes printed so far to that address (usually used to calculate output spacing)
    - `printf("item %d:%n", 3, &val)` stores 7 in `val`
    - `printf("item %d:%n", 987, &val)` stores 9 in `val`
  - `printf("000%n")`: writes the value 3 to the integer pointed to by address located 8 bytes above the RIP of `printf`

Format string vulnerabilities – walkthrough

```
char buf[64];

void vulnerable(void) {
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

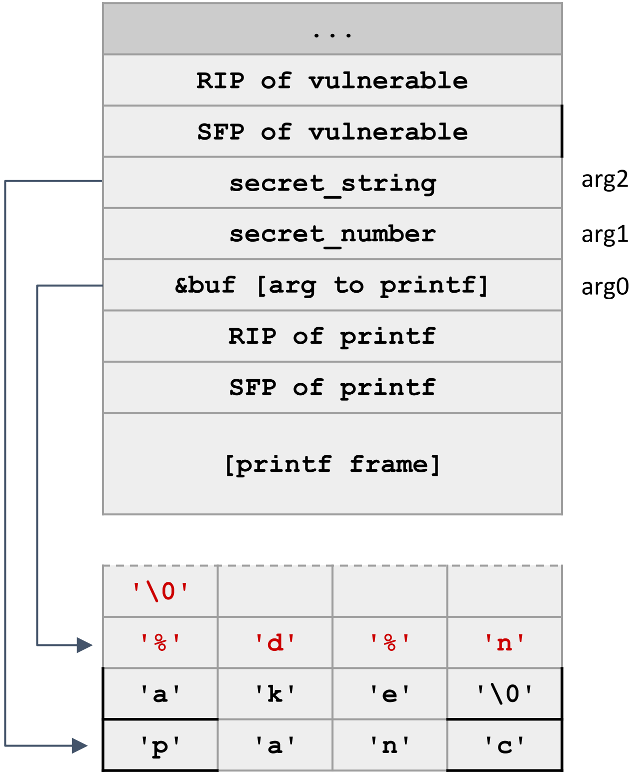
We're calling **`printf("%d%n")`**. **`printf`** reads its first argument (arg0), sees two format specifiers, and expects two more arguments (arg1 and arg2).



Format string vulnerabilities – walkthrough

```
char buf[64];

void vulnerable(void) {
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

The first format specifier `%d` says to treat the next argument (arg1) as an integer and print it out.

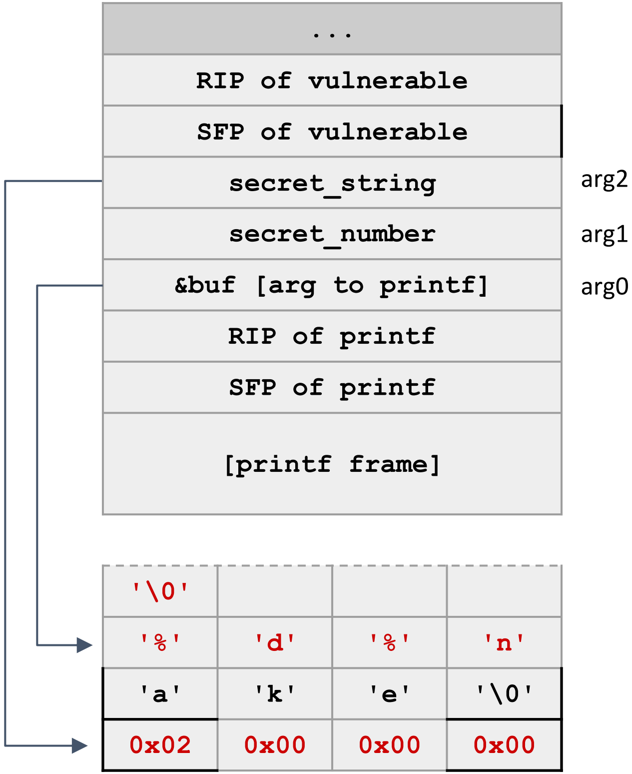


Format string vulnerabilities – walkthrough

```
char buf[64];

void vulnerable(void) {
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

The 2nd format specifier `%n` says to treat the next argument (arg2) as a pointer, and write the # of bytes printed so far to the address at arg2.

We've printed 2 bytes so far, so the number 2 gets written to **`secret_string`**.



Format string vulnerabilities – stack diagram

```
void vulnerable(void) {
    char buf[16];
    char str[12];
    fgets(buf, 28, stdin);
    printf(buf);
}
```

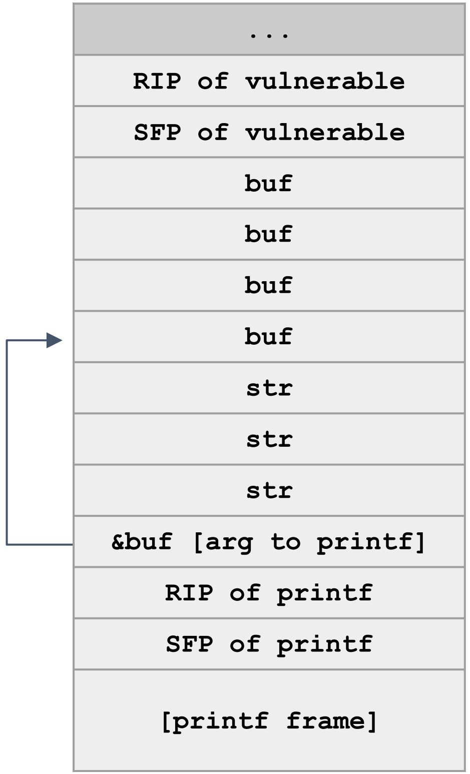
Now, let's try some format string vulnerabilities where the user-controlled buffer is on the stack instead of in static memory.

What does the stack diagram look like?

Format string vulnerabilities – stack diagram

```
void vulnerable(void) {
    char buf[16];
    char str[12];
    fgets(buf, 28, stdin);
    printf(buf);
}
```

This is the stack diagram while **printf** is being called. Where does **printf** look for arguments?

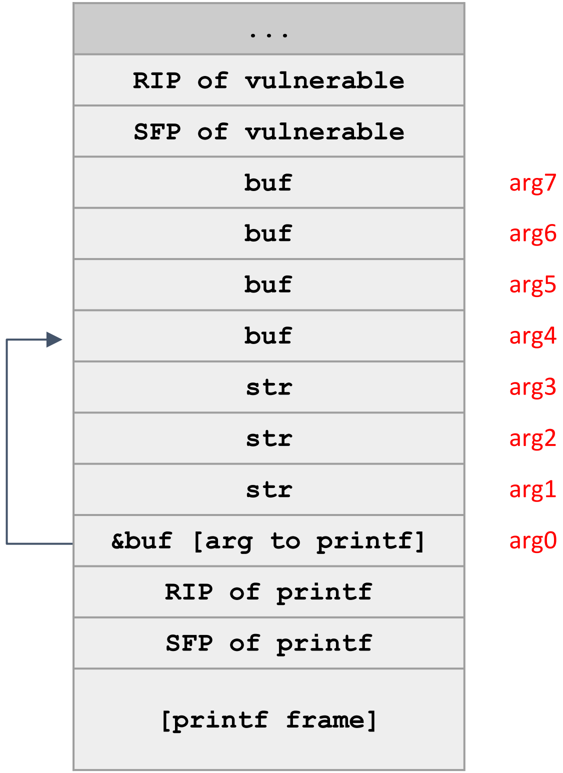


Format string vulnerabilities – stack diagram

```
void vulnerable(void) {
    char buf[16];
    char str[12];
    fgets(buf, 28, stdin);
    printf(buf);
}
```

The labels show which values in memory **`printf`** will interpret as arguments.

If **`buf`** has 4 percent formatters, **`printf`** will match the last percent formatter with arg4.



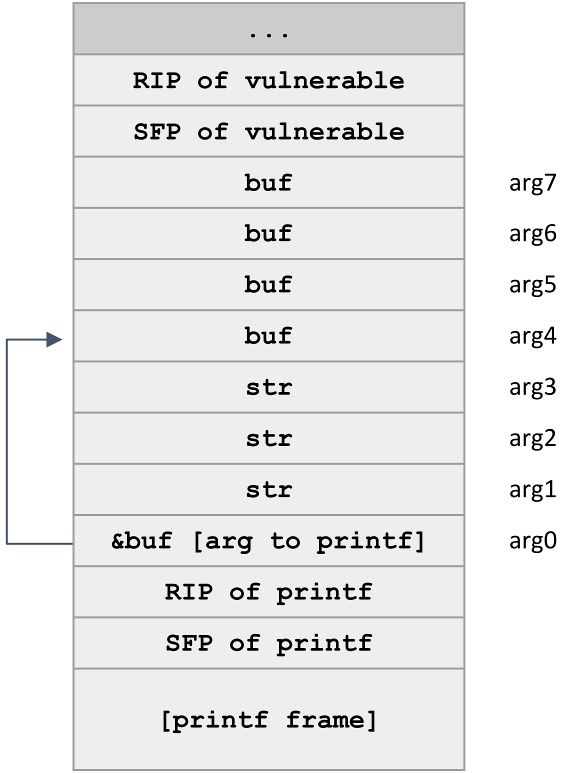
Format string vulnerabilities – write 100 to 0xdeadbeef

```
void vulnerable(void) {
    char buf[16];
    char str[12];
    fgets(buf, 28, stdin);
    printf(buf);
}
```

Recall: If **`printf`** sees a **`%n`**, it takes the next unused argument, treats it like an addr., and writes *the # of bytes printed so far* to that addr.

(1) Control *where* we write: the next unused argument on the stack is **`0xdeadbeef`**.

(2) Control *what* we write: *the # of bytes printed so far* should be 100

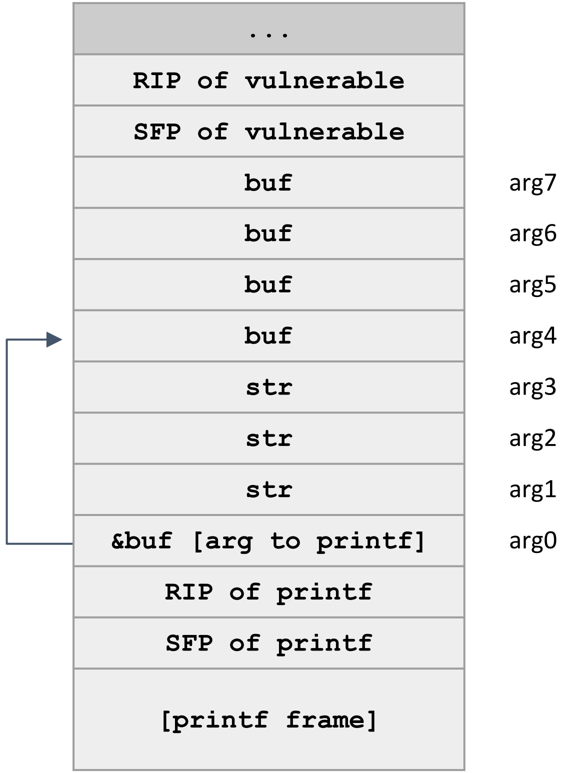


Format string vulnerabilities – write 100 to 0xdeadbeef

```
void vulnerable(void) {
     char buf[16];
     char str[12];
     fgets(buf, 28, stdin);
     printf(buf);
}
```

(1) Control *where* we write: the next unused argument on the stack is **`0xdeadbeef`**.
(2) Control *what* we write: *the # of bytes printed so far* should be 100.

Buf	0xdeadbeef	%94c	%c	%c	%n

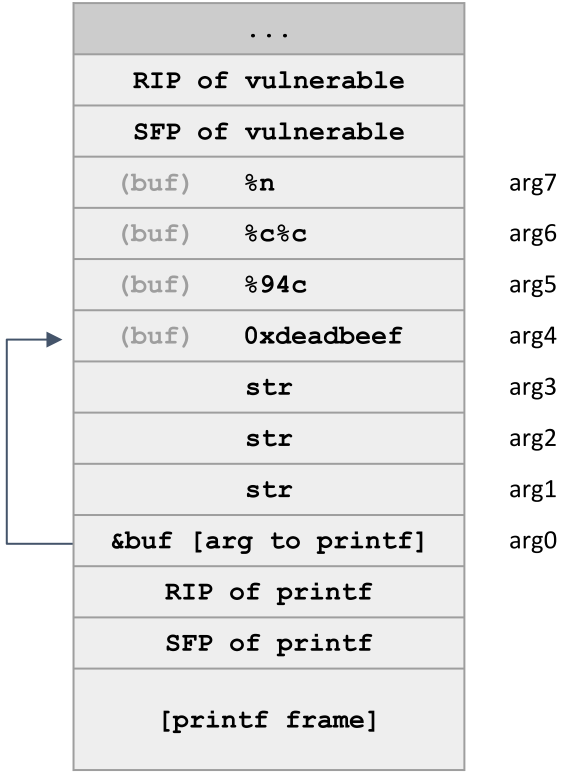


Format string vulnerabilities – write 100 to 0xdeadbeef

```
void vulnerable(void) {
    char buf[16];
    char str[12];
    fgets(buf, 28, stdin);
    printf(buf);
}
```

If we write to memory, % formatters take up multiple bytes of memory, e.g., `%94c` is 4 characters and takes up 4 bytes of memory

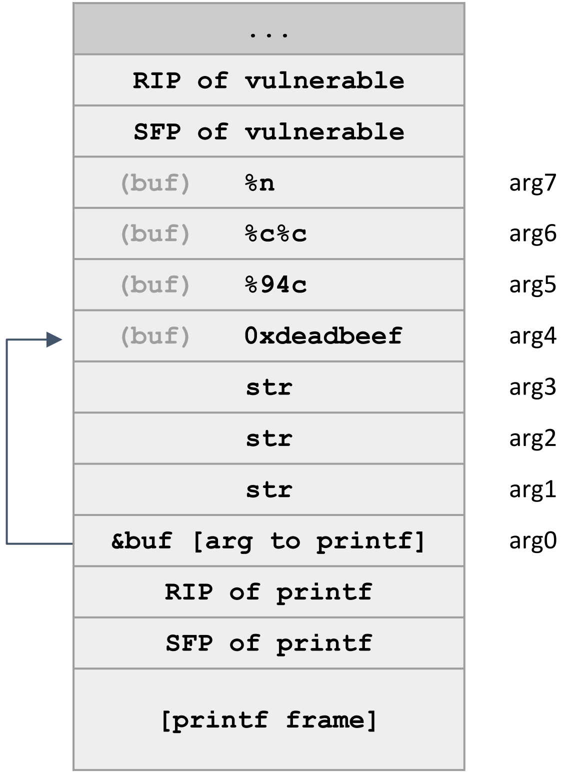
Buf	0xdeadbeef	%94c	%c	%c	%n
# Char	4	4	2	2	2



Format string vulnerabilities – write 100 to 0xdeadbeef

Control *where* we write: the next unused arg. on the stack should be **`0xdeadbeef`**.

- Each % formatter "uses up" or "consumes" one argument on the stack
- We added **`%c`** arguments to "consume" or "skip past" **`str`**, so the **`%n`** argument aligns with arg4, where we put **`0xdeadbeef`**



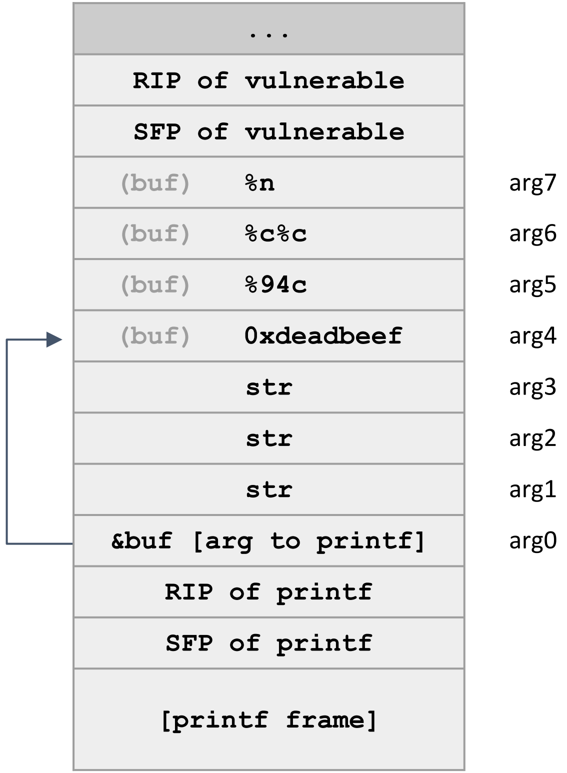
Buf	`0xdeadbeef`	`%94c`	`%c`	`%c`	`%n`
# Char	4	4	2	2	2
Args	None	arg1	arg2	arg3	arg4

Format string vulnerabilities – write 100 to 0xdeadbeef

Control *what* we write: *the # of bytes printed so far* should be 100

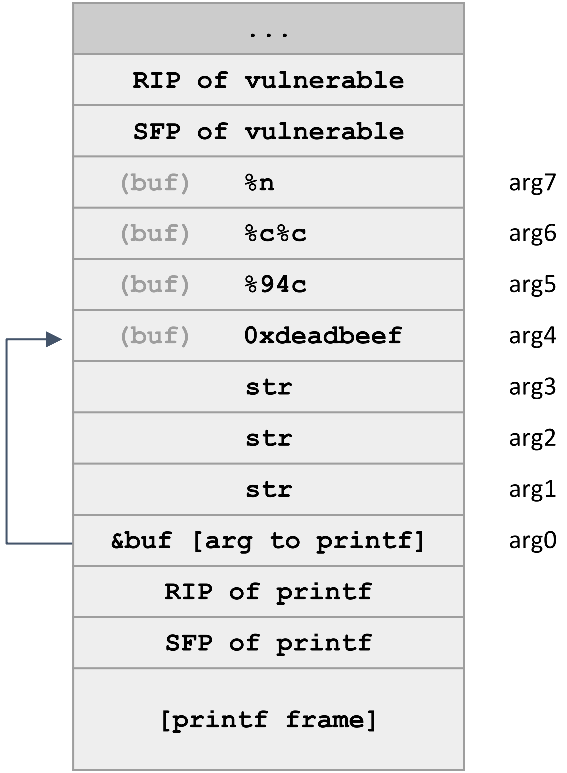
- **`%94c`** prints the next argument on the stack as a character, padded to 94 bytes (also works if you switch 94 with other numbers)
- **`0xdeadbeef`** and the **`%c`** formatters also caused characters to be printed, so we needed 100–4–1–1 = 94 padding bytes

Buf	0xdeadbeef	%94c	%c	%c	%n
# Char	4	4	2	2	2
Args	None	arg1	arg2	arg3	arg4
Print	4	94	1	1	0



Format string vulnerabilities – write 100 to 0xdeadbeef

Questions:
(1) How would you modify this exploit to write 89 bytes instead of 100 bytes?



Buf	0xdeadbeef	%94c	%c	%c	%n
# Char	4	4	2	2	2
Args	None	arg1	arg2	arg3	arg4
Print	4	94	1	1	0

Format string vulnerabilities – defense

```
void vulnerable(void) {
    char buf[64];
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf("%s", buf);
}
```

Never use untrusted input in the 1st argument to **`printf`**. Now the attacker cannot make the number of arguments mismatched!

Topics for today

- Software security
  - Motivation
  - Memory safety vulnerabilities
    - Buffer overflow vuln.
    - Integer overflow vuln.
    - Format string vuln.
    - Heap vuln.
    - Off-by-one vuln.
  - Practices to reduce software vulnerabilities

Heap vulnerabilities

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

Targeting instruction pointers

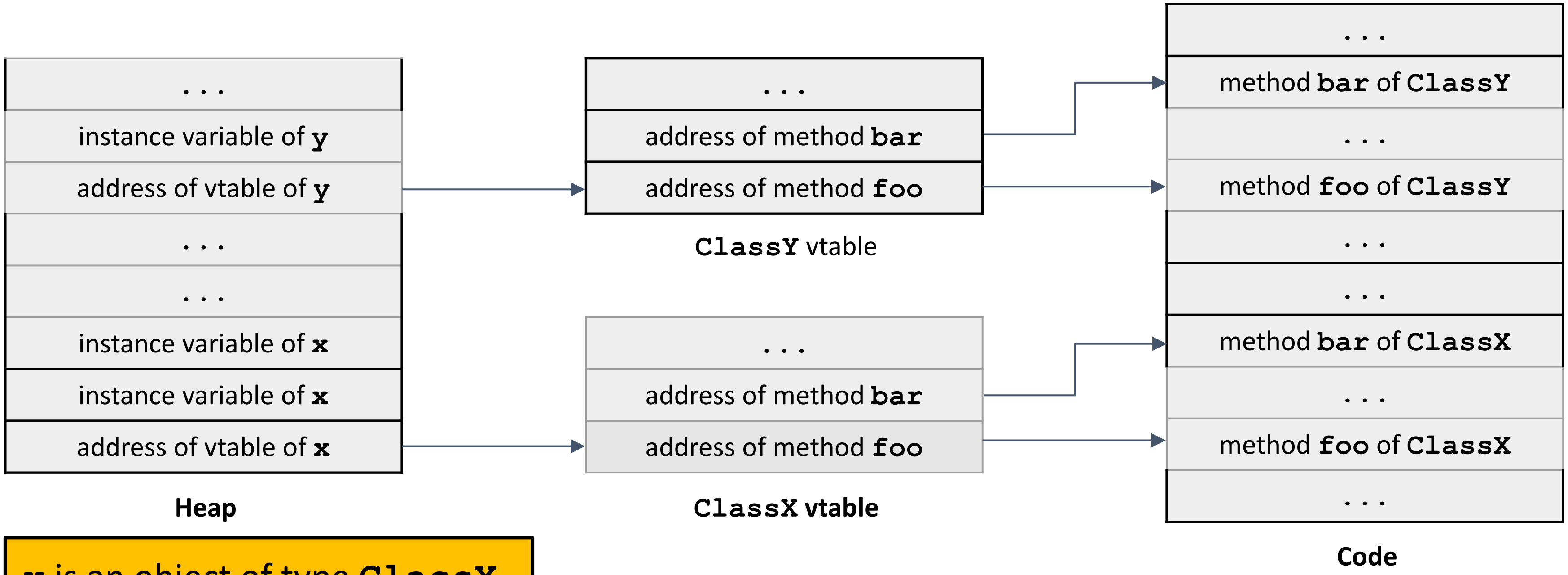
- Reminder:
  - You need to overwrite a pointer that will eventually be jumped to
  - Stack smashing controls the RIP, but there are other targets too, e.g., function pointers

C++ VTABLES

- C++ is an object-oriented language
  - C++ objects can have instance variables and methods
  - C++ has polymorphism: implementations of an interface can implement functions differently, like Java

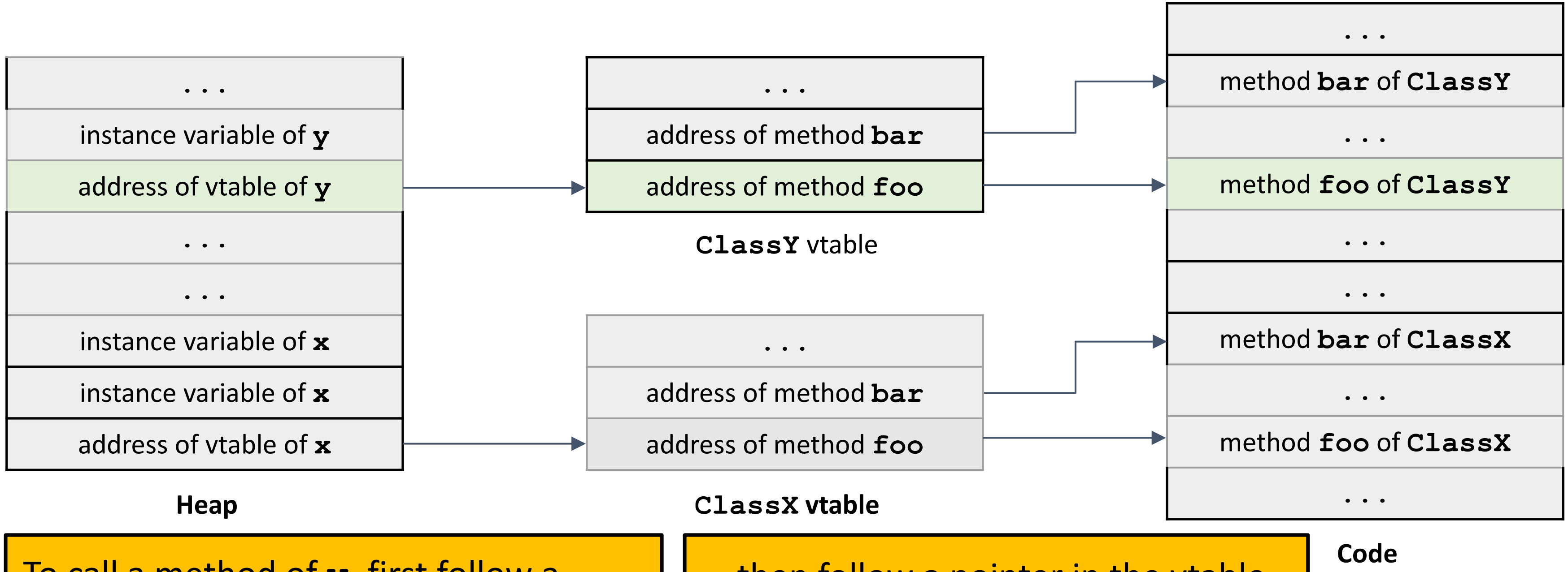
- To support this:
  - Each class has a vtable (table of fn pointers), and each object points to its class's vtable
  - The vtable pointer is usually at the beginning of the object
  - To run a fn: dereference the vtable pointer with an offset to find the function address

C++ VTABLES



x is an object of type **ClassX**.
y is an object of type **ClassY**.

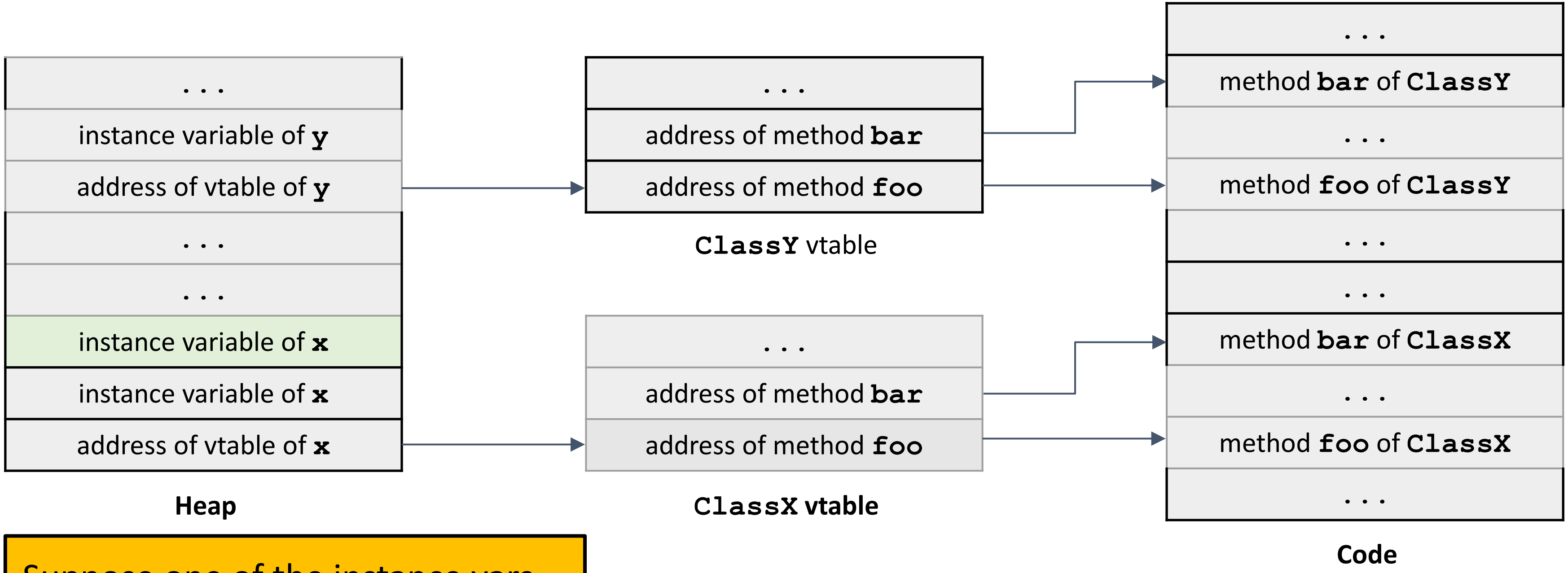
C++ VTABLES



To call a method of **y**, first follow a pointer on the heap to find the vtable…

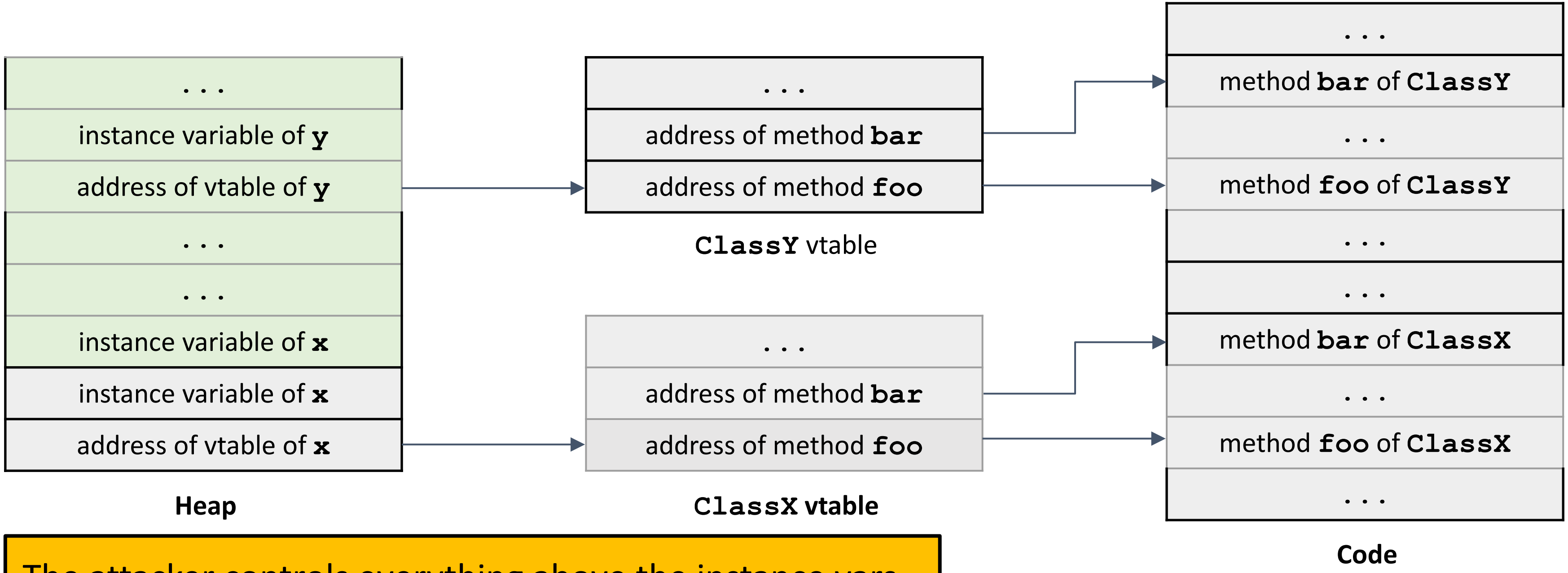
… then follow a pointer in the vtable to find the instructions of the method

C++ VTABLES



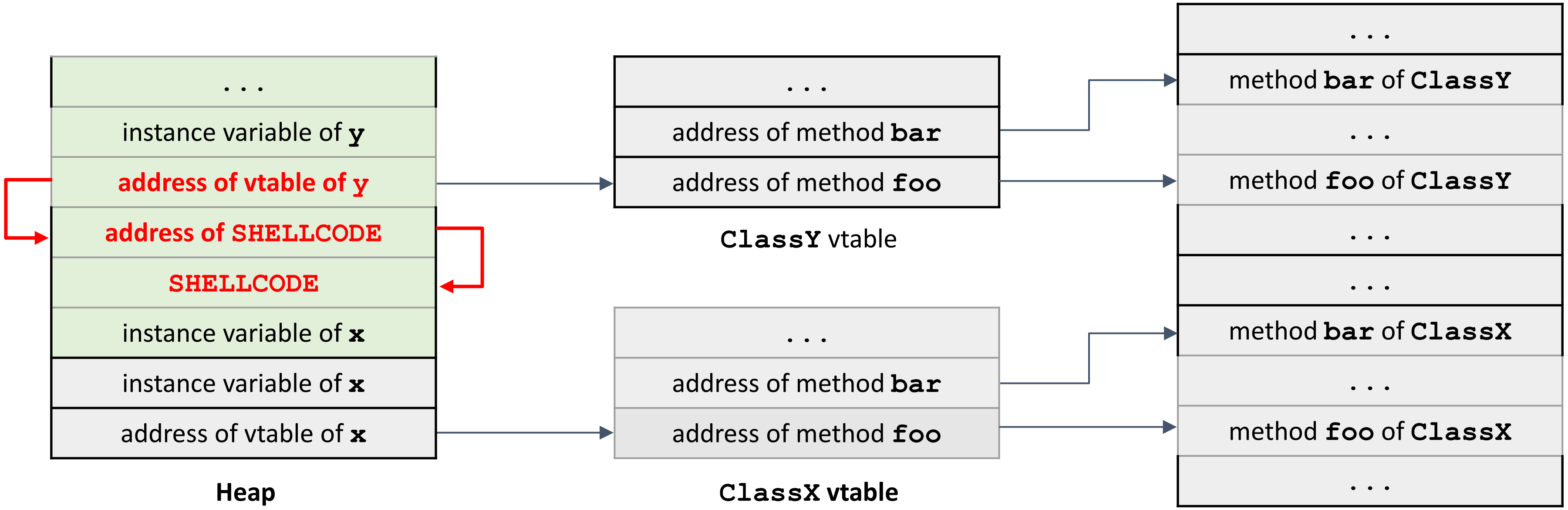
Suppose one of the instance vars. of **x** is a buffer we can overflow

C++ VTABLES



The attacker controls everything above the instance vars. of **x** on the heap, including the vtable pointer for **y**

C++ VTABLES



The vtable for `y` is now a pointer to shellcode. If method `foo` for `y` is called, it will execute shellcode!

Heap vulnerabilities

- Heap overflow
  - Objects are allocated in the heap (using `malloc` in C or `new` in C++)
  - A write to a buffer in the heap is not checked
  - The attacker overflows the buffer and overwrites the vtable pointer of the next object to point to a malicious vtable, with pointers to malicious code
  - The next object's function is called, accessing the vtable pointer

- Use-after-free
  - An object is deallocated too early (using `free` in C or `delete` in C++)
  - The attacker allocates memory, which returns the memory freed by the object
  - The attacker overwrites a vtable pointer under the attacker's control to point to a malicious vtable, with pointers to malicious code
  - The deallocated object's function is called, accessing the vtable pointer

Heap vulnerabilities: use-after-free

- Allocate memory in func1()
  - char *m = malloc(16), put Hello, world
- Free that block in func2(m)
  - free(m)
- Allocate memory in func3()
  - char *m2 = malloc(16), put Not hello, world
- Use m in func4 (?!)

```
char * func1() {
    char *m = malloc(16);
    strncpy(m, "Hello world", 16);
    return m;
}

void func2(char *m) {
    free(m);
}

char * func3() {
    char *m2 = malloc(16);
    strncpy(m2, "Not Hello world", 16);
    return m2;
}

void func4(char *m) {
    printf("%s\n", m);
}

int main() {
    char *m = func1();
    func2(m);
    func3();
    func4(m);
}
```

Topics for today

- Software security
  - Motivation
  - Memory safety vulnerabilities
    - Buffer overflow vuln.
    - Integer overflow vuln.
    - Format string vuln.
    - Heap vuln.
    - Off-by-one vuln.
  - Practices to reduce software vulnerabilities

Off-by-one vulnerability

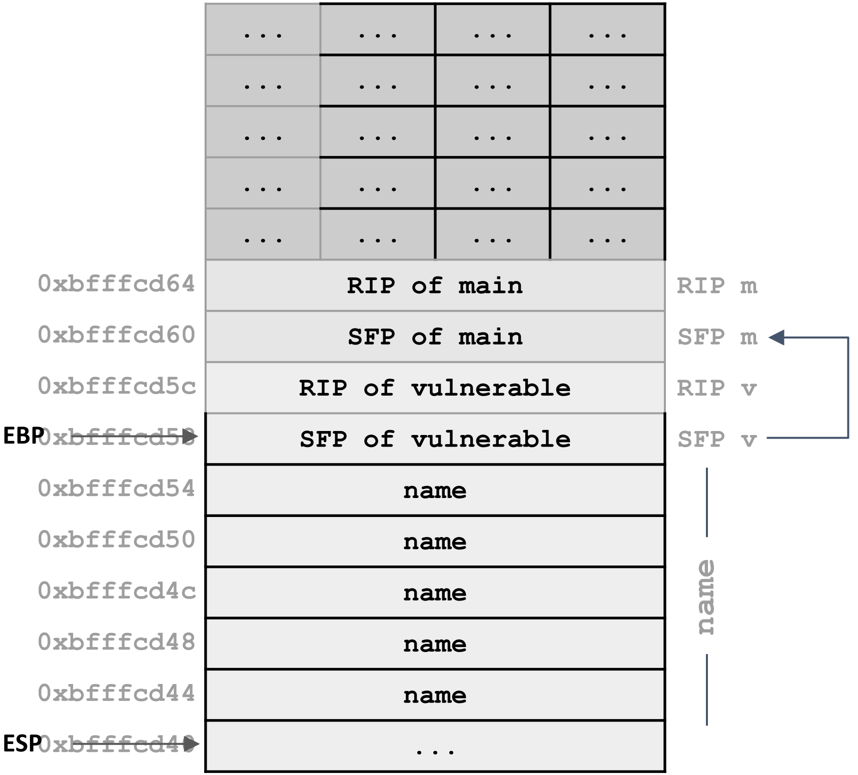
Goal: execute shellcode located at **`0xdeadbeef`**.

What parts of memory is an attacker able to overwrite in this piece of code?

```
void vulnerable(void) {
    char name[20];
    fread(name,21,1,stdin);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP →

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```

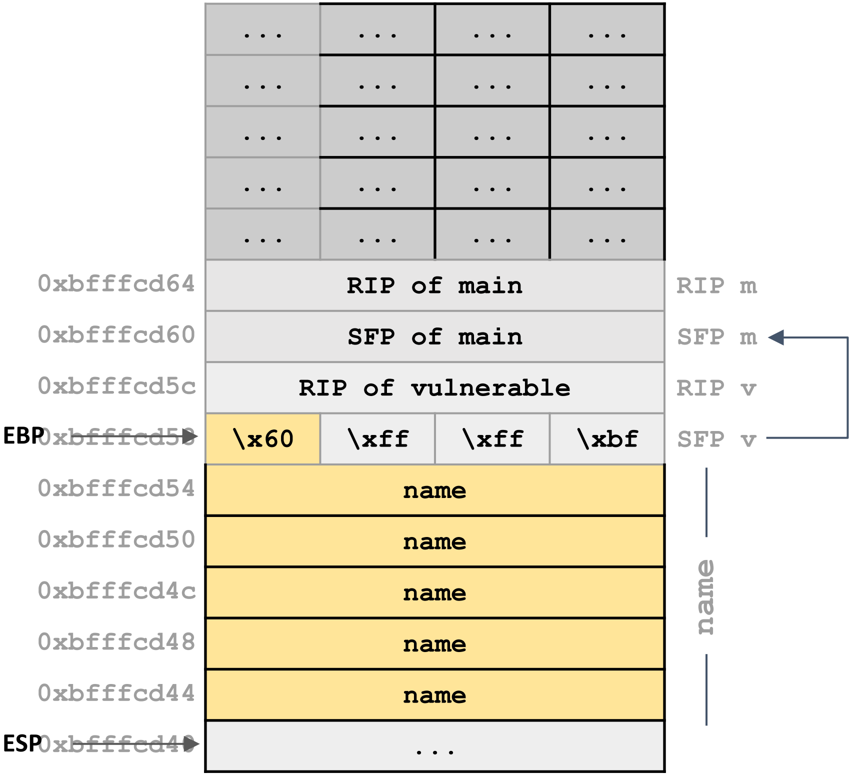


Off-by-one vulnerability – cont'd

The attacker can overwrite all of **`name`** and the least-significant byte of the SFP of **`vulnerable`**. If the attacker can change where **`vulnerable`** points, how to exploit this to execute shellcode?

```
void vulnerable(void) {
    char name[20];
    fread(name,21,1,stdin);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
vulnerable:
    ...
    call gets      <- EIP
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



Off-by-one vulnerability – cont'd

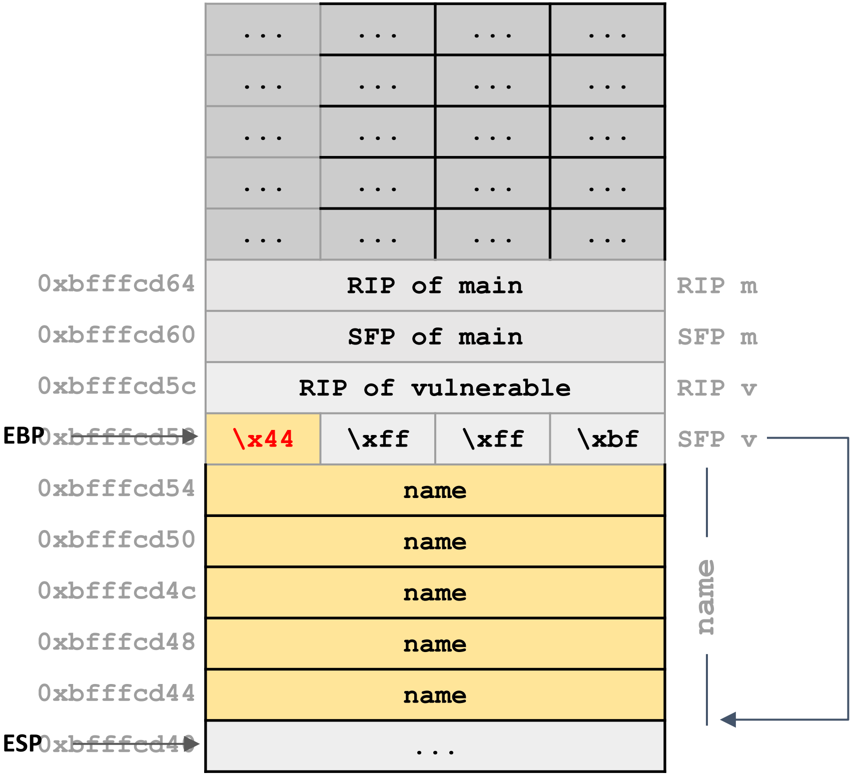
Suppose we put 0x44. The SFP of **`vulnerable`** points inside **`name`**, which the attacker controls.

What does the SFP usually point to? What will the C program interpret the first bytes of **`name`** as?

```
void vulnerable(void) {
    char name[20];
    fread(name,21,1,stdin);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP → `call gets`

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



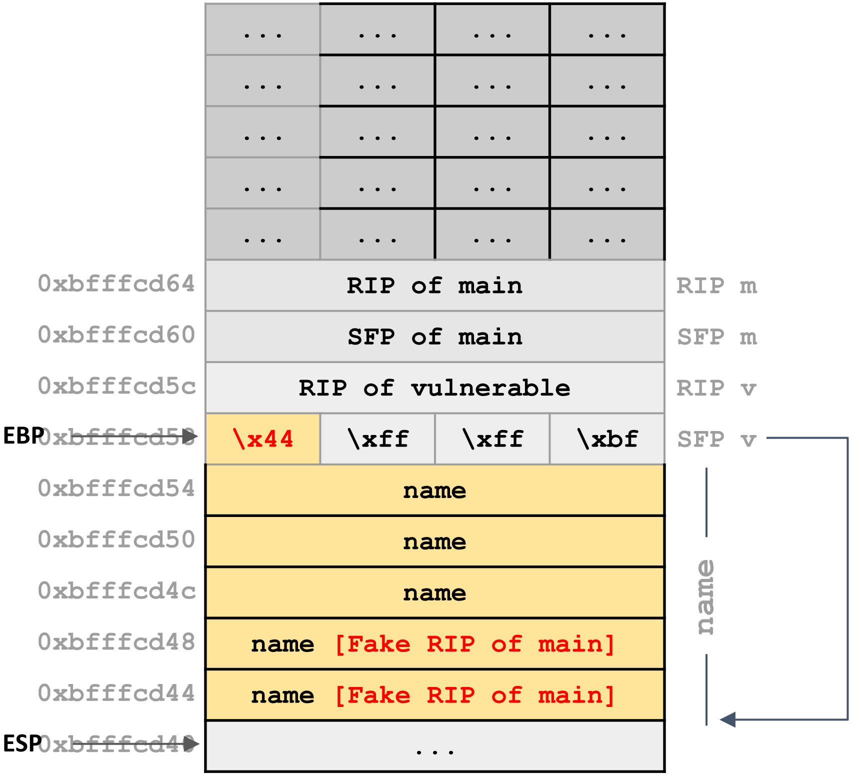
Off-by-one vulnerability – cont'd

The C program now thinks that the SFP of **`main`** and the RIP of **`main`** are inside **`name`**. The attacker controls these values, so they can now overwrite where the program thinks the RIP of **`main`** is.

```c
void vulnerable(void) {
    char name[20];
    fread(name,21,1,stdin);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP → `call gets`

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



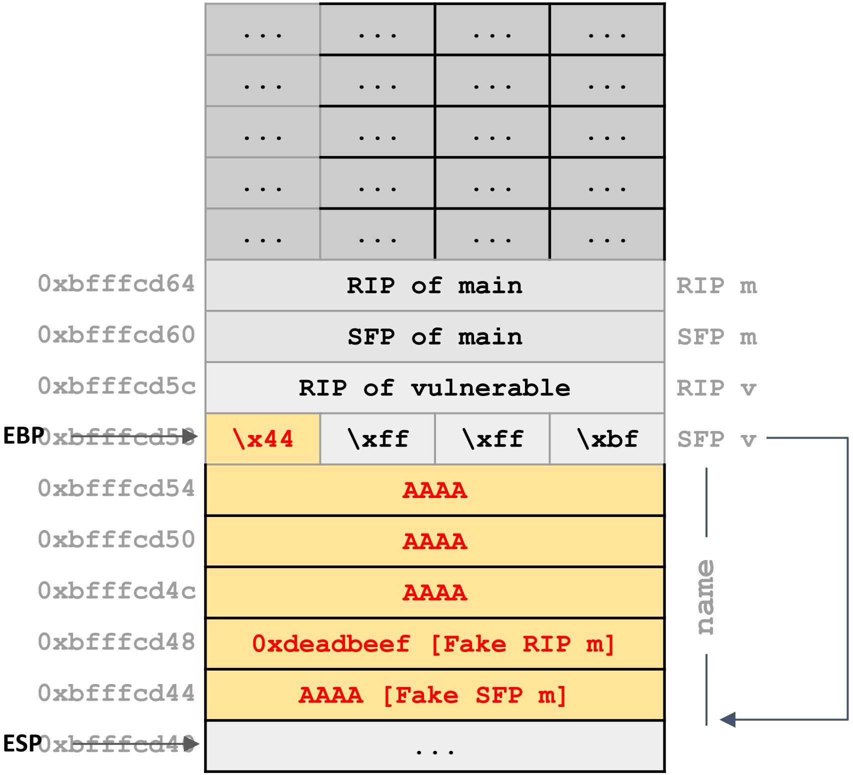
OFF-BY-ONE VULNERABILITY – CONT'D

Let's see what happens
when the **`vulnerable`** function returns.

```
void vulnerable(void) {
    char name[20];
    fread(name,21,1,stdin);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP →

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



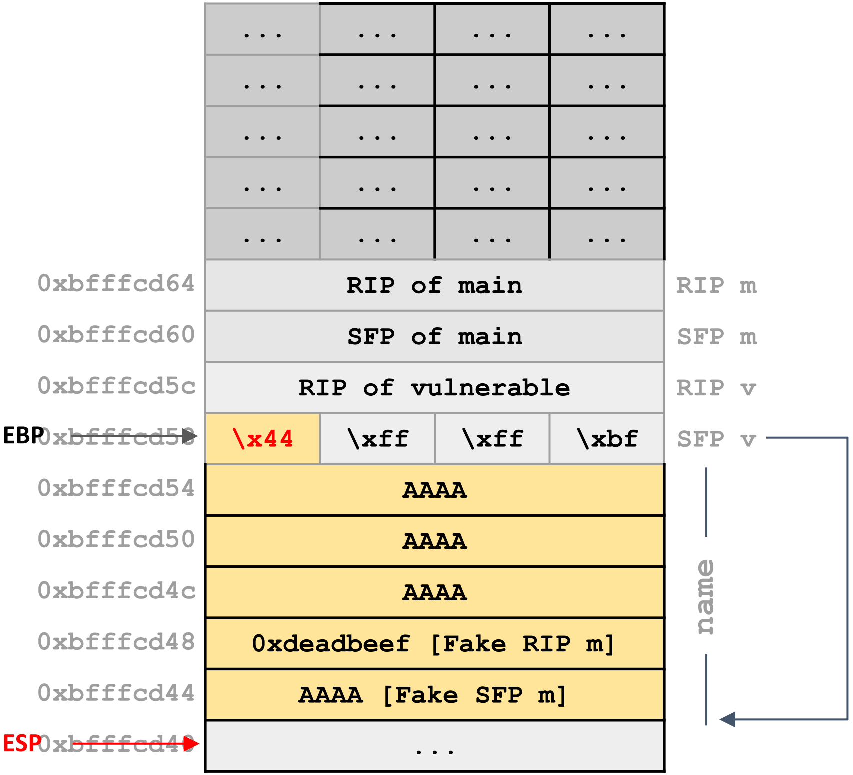
Off-by-one vulnerability – cont'd

Returning from **gets**,
preparing to return from **vulnerable**.

```
void vulnerable(void) {
    char name[20];
    fread(name,21,1,stdin);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP →

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp      <- EIP
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



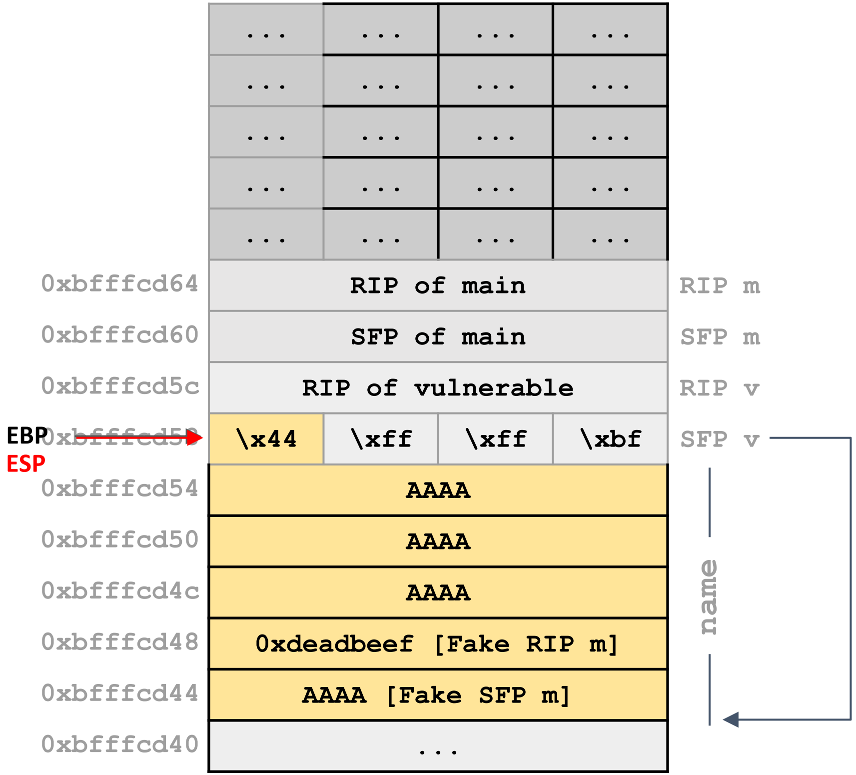
Off-by-one vulnerability – cont'd

Epilogue step 1: Move ESP back up.

```
void vulnerable(void) {
    char name[20];
    fread(name,21,1,stdin);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP → (at `pop %ebp`)

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



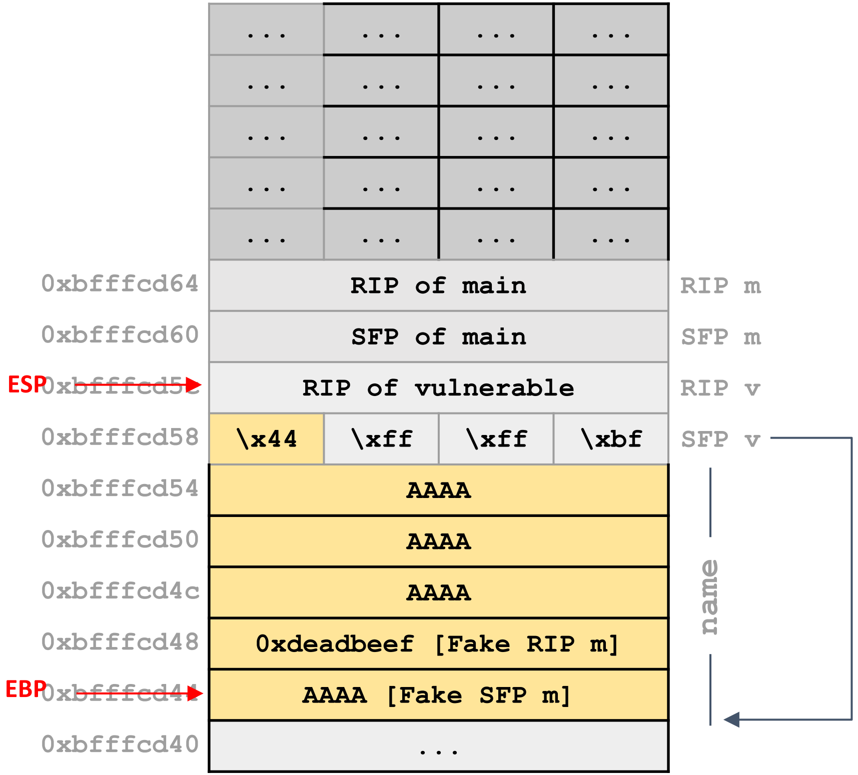
Off-by-one vulnerability – cont'd

Epilogue step 1: Move ESP back up
Epilogue step 2: Restore EBP. Note that EBP now points inside **`name`**, instead of at the SFP of **`main`**

```
void vulnerable(void) {
    char name[20];
    fread(name,21,1,stdin);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP →

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



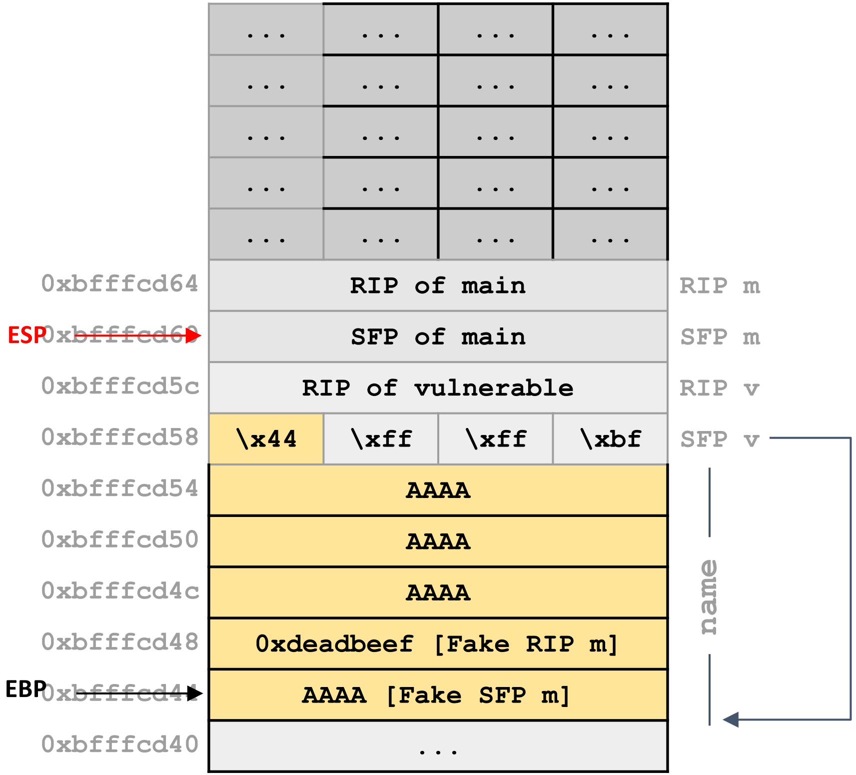
Off-by-one vulnerability – cont'd

Epilogue step 1: Move ESP back up
Epilogue step 2: Restore EBP. Note that EBP now points inside **`name`**, instead of at the SFP of **`main`**
Epilogue step 3: Restore EIP. We never changed the RIP of **`vulnerable`**, so it returns to **`main`**

```
void vulnerable(void) {
    char name[20];
    fread(name,21,1,stdin);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP →

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



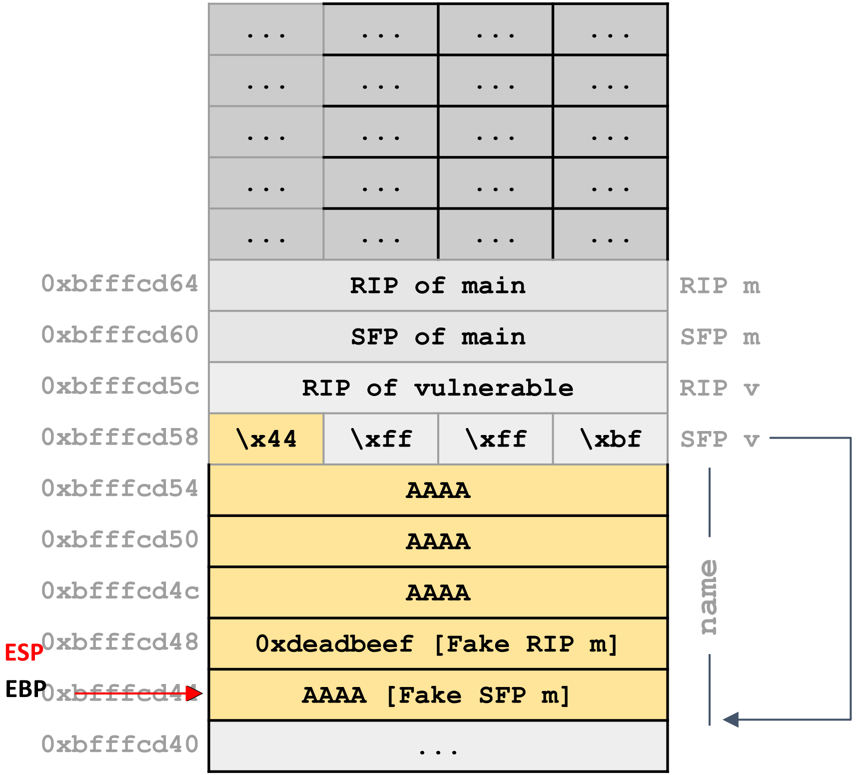
Off-by-one vulnerability – cont'd

Let's see what happens when the **main** function returns, now with the EBP in the wrong place

Epilogue step 1: Move ESP back up

```
void vulnerable(void) {
    char name[20];
    fread(name,21,1,stdin);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp       <- EIP
    ret
```



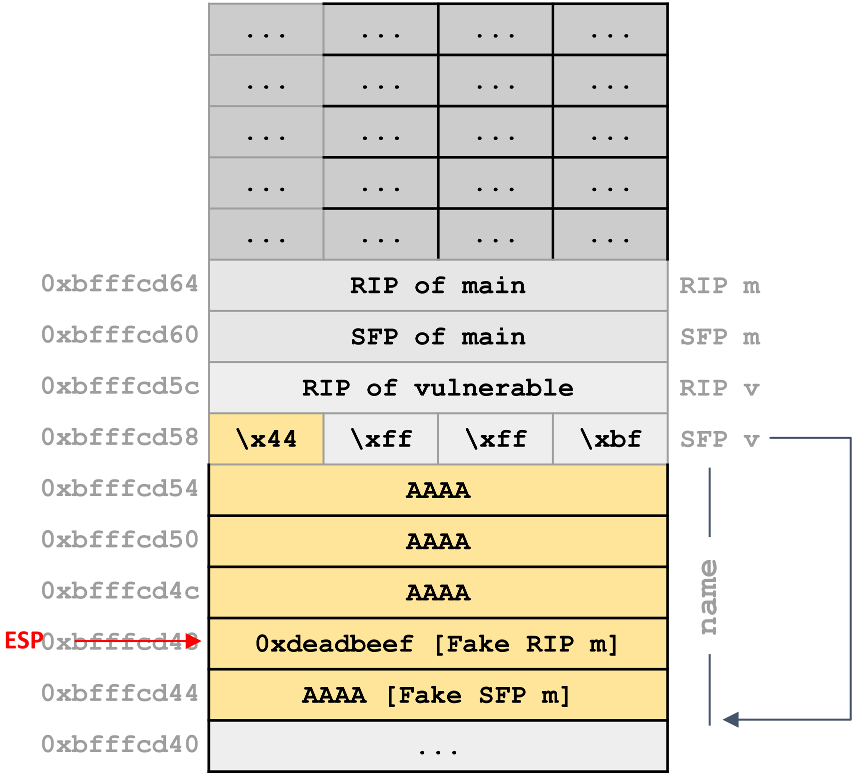
Off-by-one vulnerability – cont'd

EBP → ?

Epilogue step 1: Move ESP back up
Epilogue step 2: Restore EBP; The program looks at our fake SFP to restore EBP, and points EBP to garbage AAAA

```
void vulnerable(void) {
    char name[20];
    fread(name,21,1,stdin);
}

int main(void) {
    vulnerable();
    return 0;
}
```

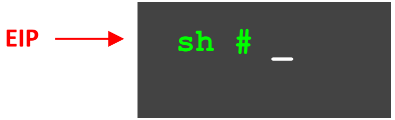
```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
EIP → ret
```



Off-by-one vulnerability – cont'd

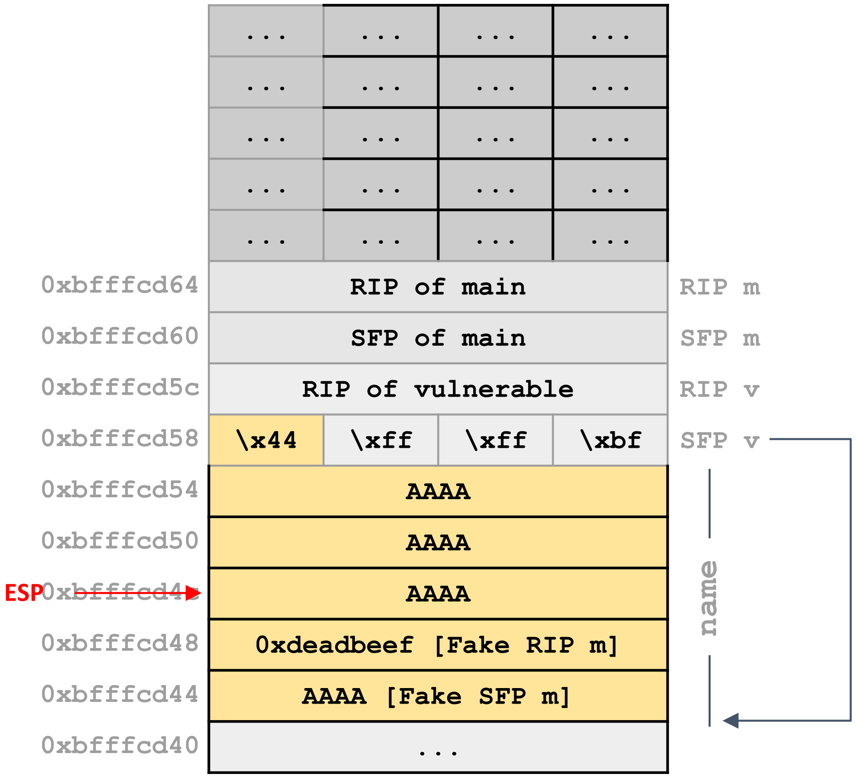
EBP → ?

Epilogue step 1: Move ESP back up
Epilogue step 2: Restore EBP
Epilogue step 3: Restore EIP. The program looks at our fake RIP to restore EIP, and redirects execution to `0xdeadbeef`




```
void vulnerable(void) {
    char name[20];
    fread(name,21,1,stdin);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



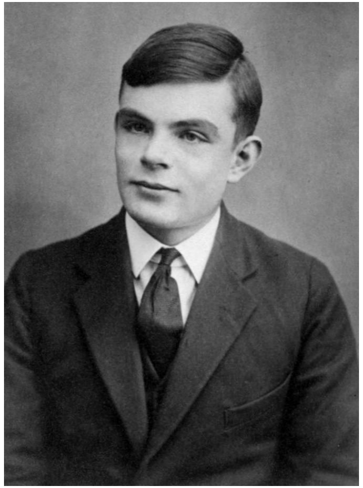
Topics for today

- Software security
  - Motivation
  - Memory safety vulnerabilities
    - Buffer overflow vuln.
    - Integer overflow vuln.
    - Format string vuln.
    - Heap vuln.
    - Off-by-one vuln.
  - Practices to reduce software vulnerabilities

Can we avoid vulnerabilities?

- Is it a solve-able problem?
  - Suppose we have code A and want to tell if it has mistakes or not
  - The code is unlimitedly large, and we have unlimited resources
  - Can't tell if the code has a vulnerability or not (`Halting Problem`)

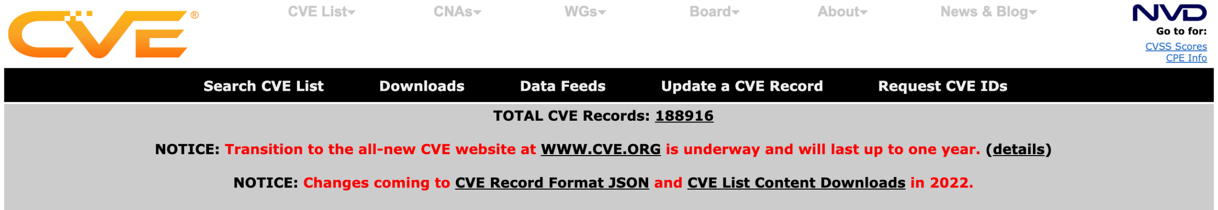
- Is it pessimistic future?
  - **No**
  - Fortunately, code has a limited size, and we have limited resources
  - Can reduce the number of mistakes in the code
    - Construct patterns of existing vulnerabilities and search those patterns (pattern matching)
    - Run the program with various inputs and find any crashes/vulnerabilities (fuzzing)
    - … (many more)



Alan Turing…

Require vulnerability database

- Common vulnerabilities and exposures



  - Maintained by NIST/MITRE
  - Software vulnerability can inflict a huge impact
  - We use this database to announce common vulnerabilities to the community

Require vulnerability database – cont'd

- How does it work?
  - Developers
    - Find vulnerabilities in their software (e.g., NGINX v1.0.7 ~ 1.0.14 has a BOF)
    - Fix them
    - Announce the fixes to CVE

Vulnerability Details : CVE-2012-2089

Buffer overflow in ngx_http_mp4_module.c in the ngx_http_mp4_module module in nginx 1.0.7 through 1.0.14 and 1.1.3 through 1.1.18, when the mp4 directive is used, allows remote attackers to cause a denial of service (memory overwrite) or possibly execute arbitrary code via a crafted MP4 file.

Publish Date : 2012-04-17 Last Update Date : 2021-11-10

  - System operators
    - Watch the CVE list and update vulnerable software

Require vulnerability database – cont'd

- How does it work?
  - White hat hackers
    - Analyze software using testing methods
    - Fuzzing, symbolic execution, manual testing, code auditing, reverse engineering, etc
    - Find a bug
    - Exploit the bug
  - Vendors
    - Run bug bounty programs
    - Vendor reports the vulnerabilities white-hat hackers found to NIST/MITRE CVE

- **syslog**

  Available for: iPhone 4s and later, iPod touch (5th generation) and later, iPad 2 and later

  Impact: A local user may be able to change permissions on arbitrary files

  Description: syslogd followed symbolic links while changing permissions on files. This issue was addressed through improved handling of symbolic links.

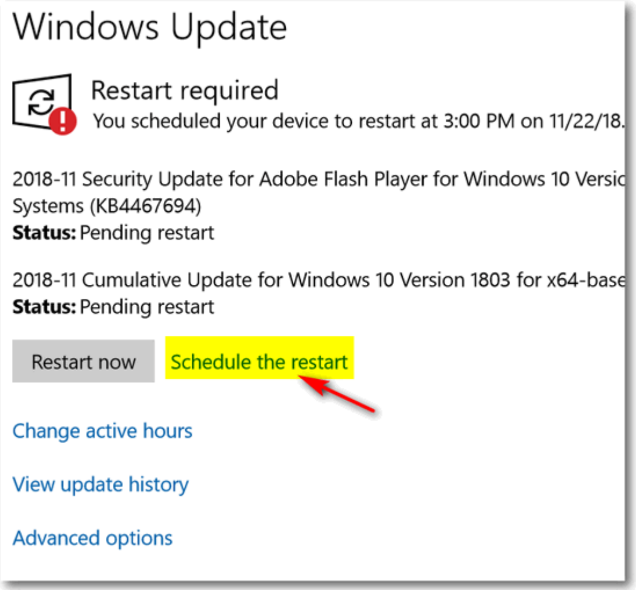
  CVE-ID

  CVE-2014-4372 : Tielei Wang and YeongJin Jang of Georgia Tech Information Security Center (GTISC)

Users need to patch software immediately

- Facts
  - Vulnerabilities are reported every day
  - We cannot fix all the vulnerabilities at once (it requires testing, testing, testing…)

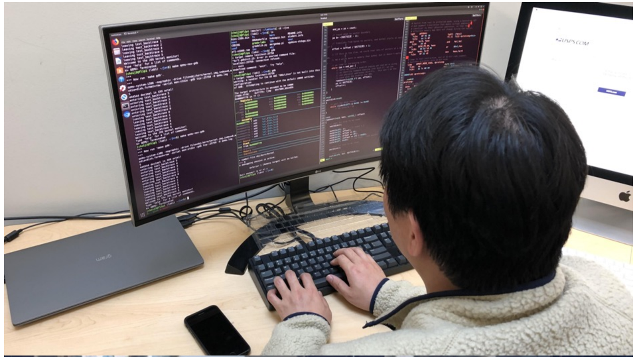
- Recommendations
  - Do not miss the updates
  - Developers set patch schedules
    - MS Windows regularly issues a patch on 2nd Tue.
  - Missing them gives opportunities to hackers..



Help developers reduce mistakes

- Unit tests
  - Create test-cases and run before committing your code

- Do code review
  - Put a non-stressful human here
  - They will read code in a different perspective



Topics for today

- Software security
  - Motivation
  - Memory safety vulnerabilities
    - Buffer overflow vuln.
    - Integer overflow vuln.
    - Format string vuln.
    - Heap vuln.
    - Off-by-one vuln.
  - Practices to reduce software vulnerabilities

Thank You!

Tu/Th 4:00 – 5:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab