extended disc. 2

cs161 su23

welcome!

X86, memory safety vulnerabilities

slides

bit.ly/cs161-disc

feedback

bit.ly/extended-feedback

about me — abhi

- abhi (he/him/his)
- from st. louis, missouri
- love writing and film photography (recently)
- i’m here to be your point of contact!
  - 1-hr disc: M/W 5-6pm Wheeler 202
  - abhiganesh@berkeley.edu

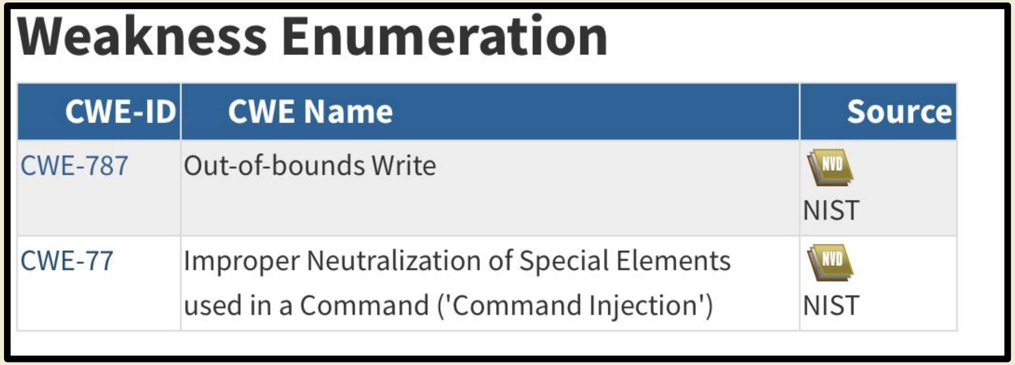
hack of the day

hack of the day

- RealTek Jungle SDK vulnerability led to 134 million IOT device exploit attempts

hack of the day

- RealTek Jungle SDK vulnerability led to 134 million IOT device exploit attempts
  - done through buffer overflow injecting shellcode

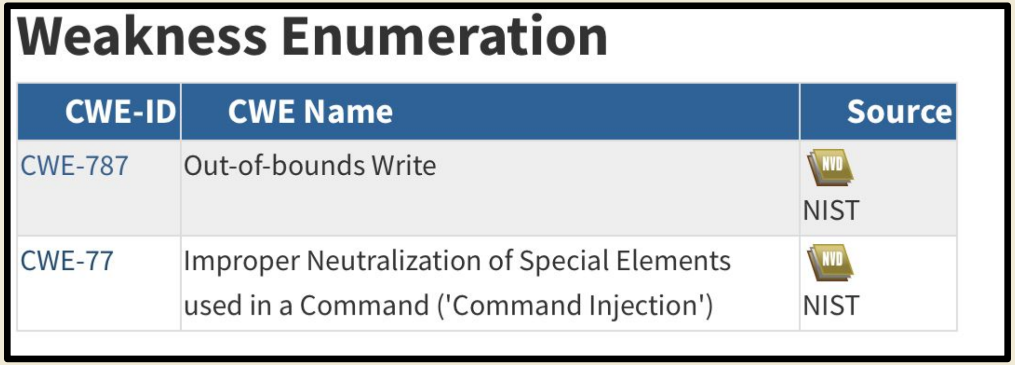


Weakness Enumeration

CWE-ID	CWE Name	Source
CWE-787	Out-of-bounds Write	NVD NIST
CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	NVD NIST

hack of the day

- RealTek Jungle SDK vulnerability led to 134 million IOT device exploit attempts
  - done through buffer overflow injecting shellcode
  - IOT devices potentially executed malware



Weakness Enumeration

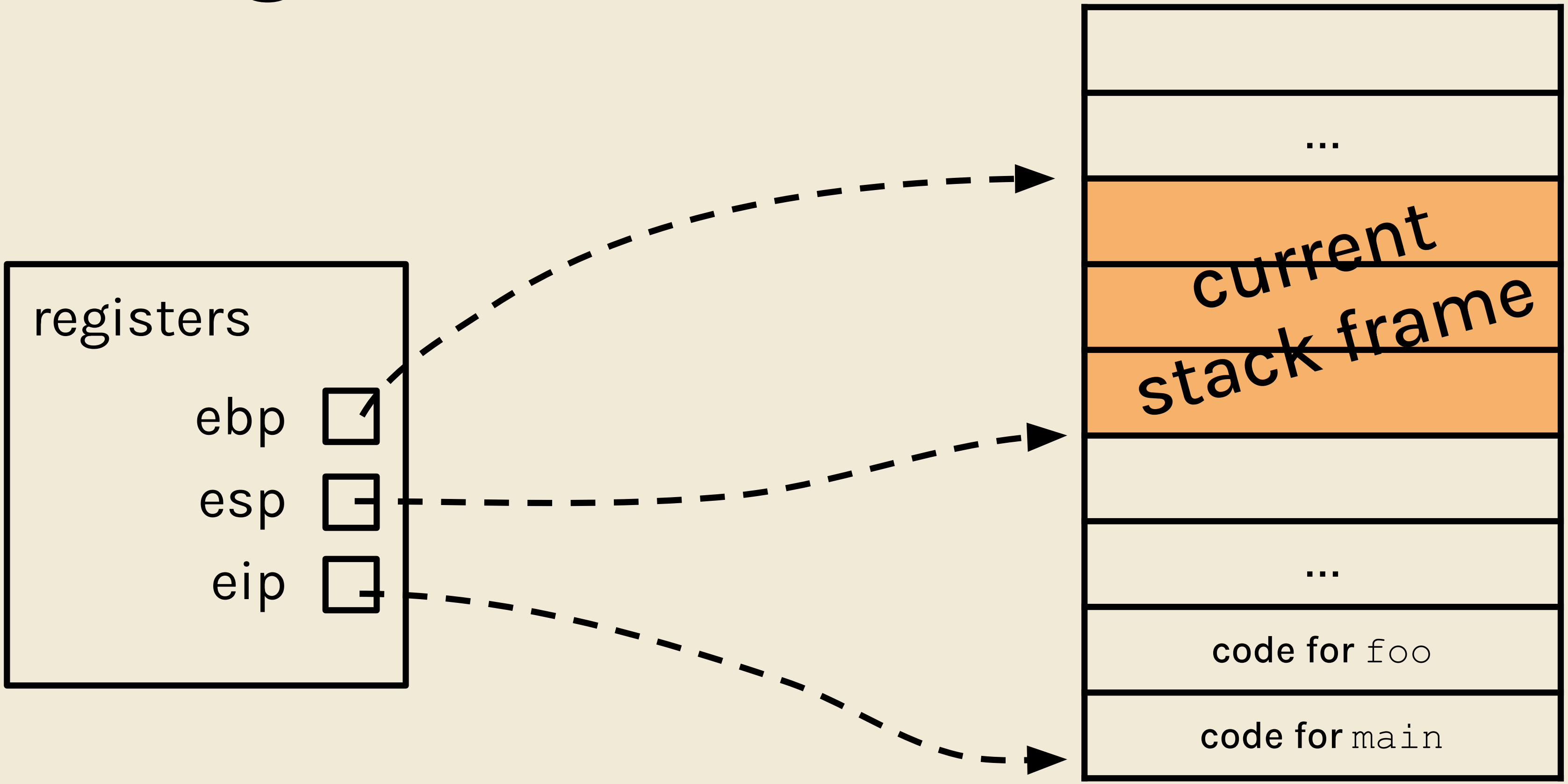
CWE-ID	CWE Name	Source
CWE-787	Out-of-bounds Write	NVD NIST
CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	NVD NIST

general questions, concerns, etc.

X86 review

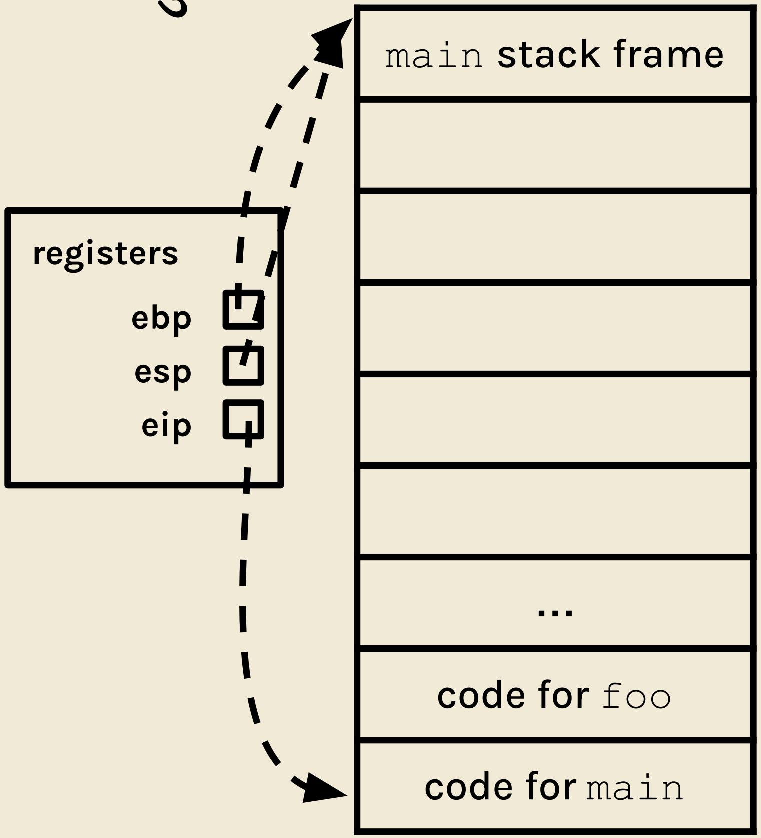
no, it’s not RISC-V

the registers



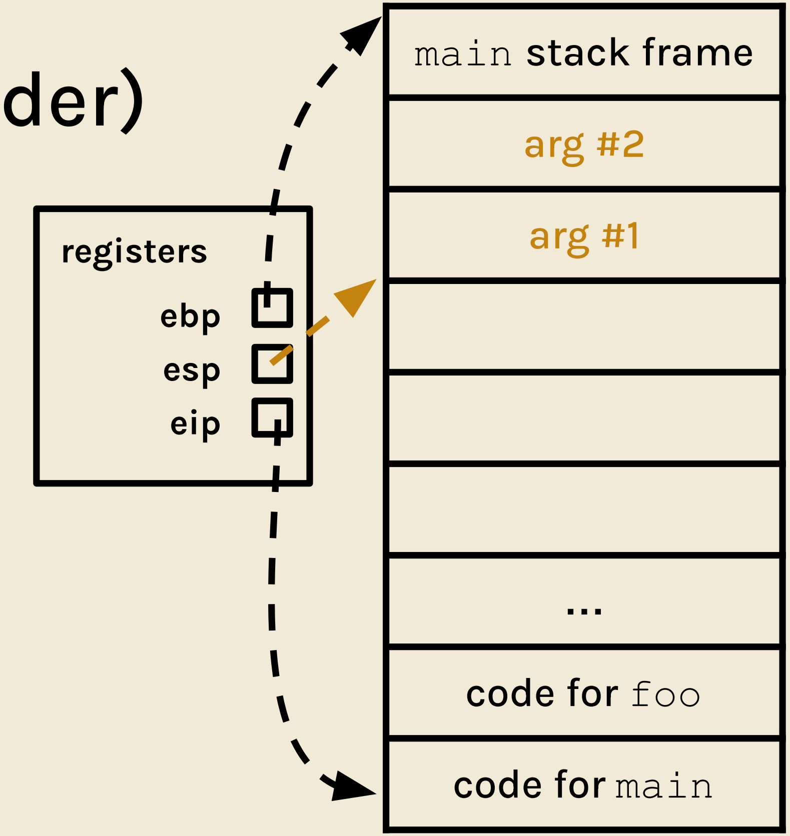
calling convention

```
main() {
    foo(arg1, arg2)
}
```



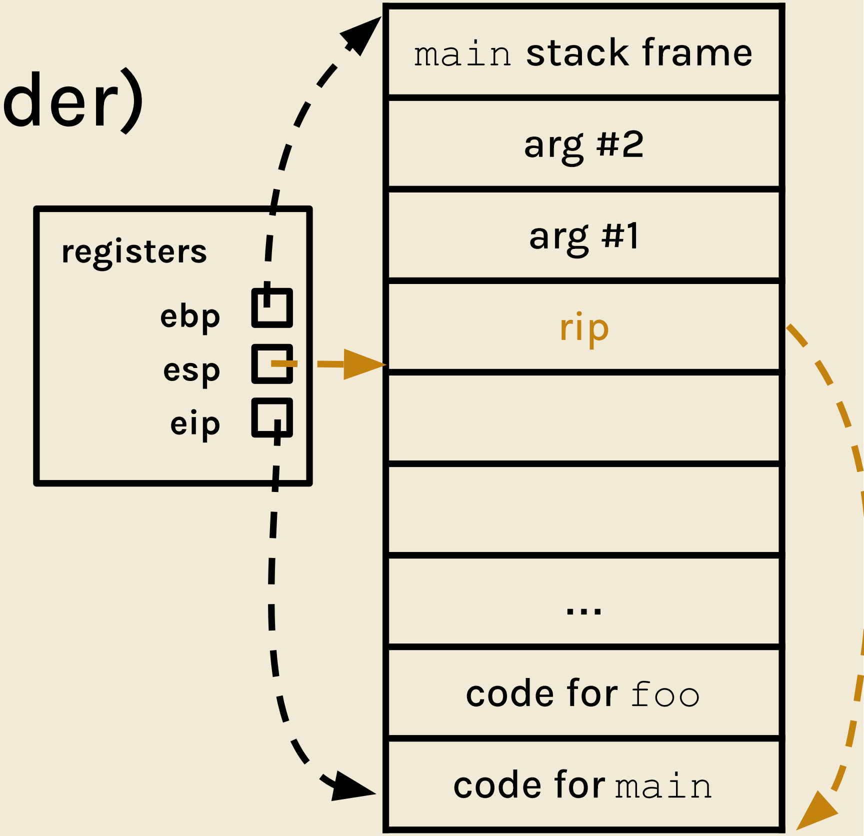
calling convention

1. push arguments (reverse order)
   - adjust esp



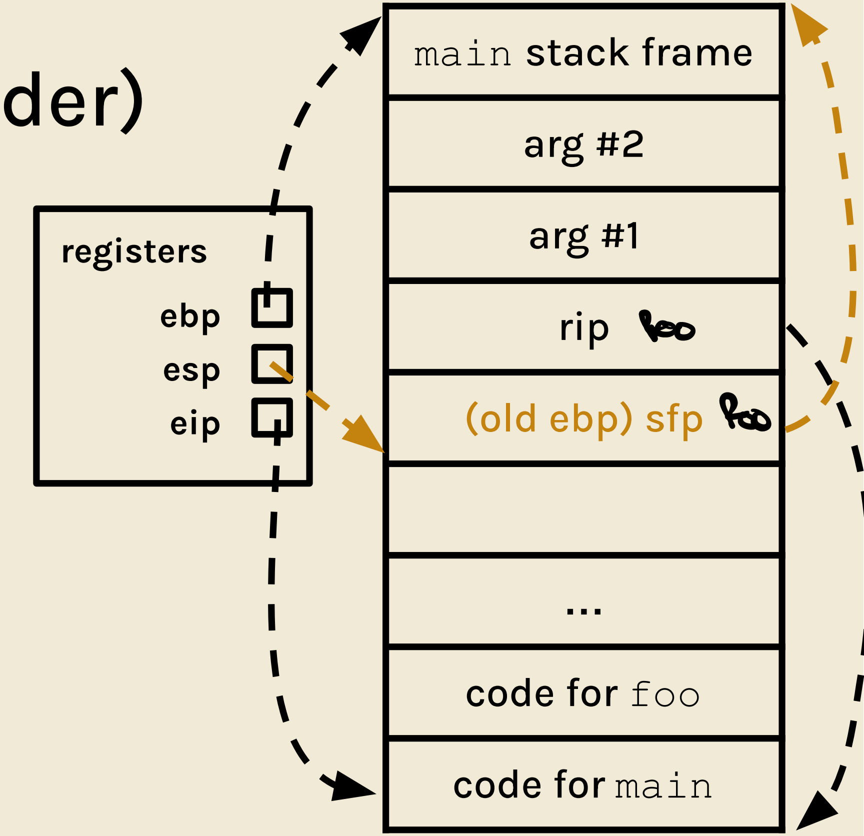
calling convention

1. push arguments (reverse order)
2. remember eip
   - like `ra` in RISC-V



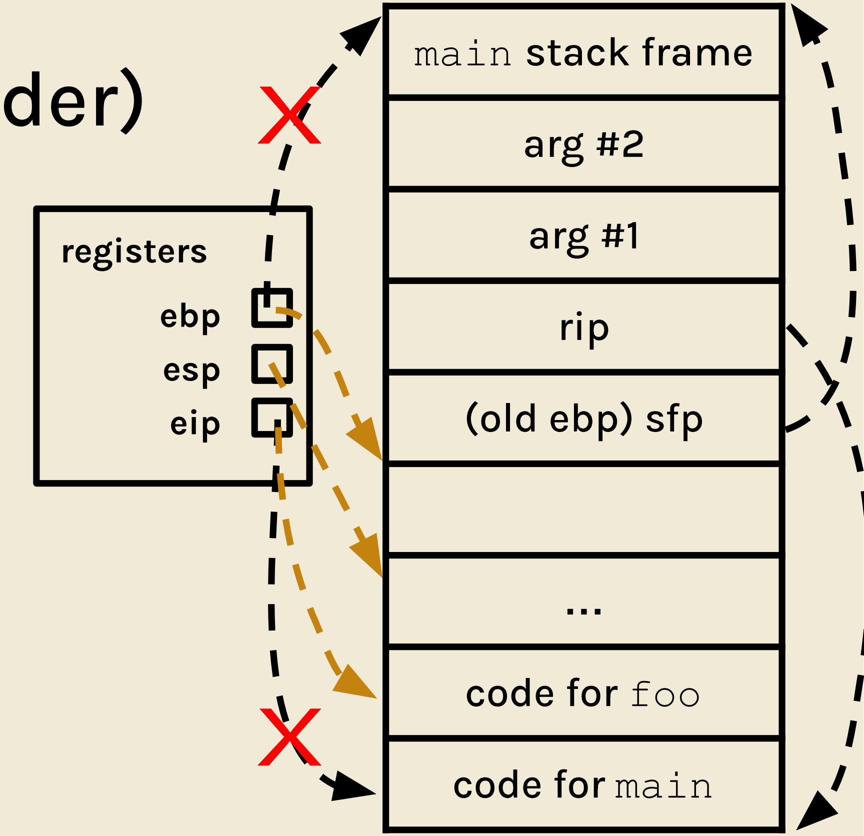
calling convention

1. push arguments (reverse order)
2. remember eip
3. remember ebp
   - to restore to top of previous stack frame



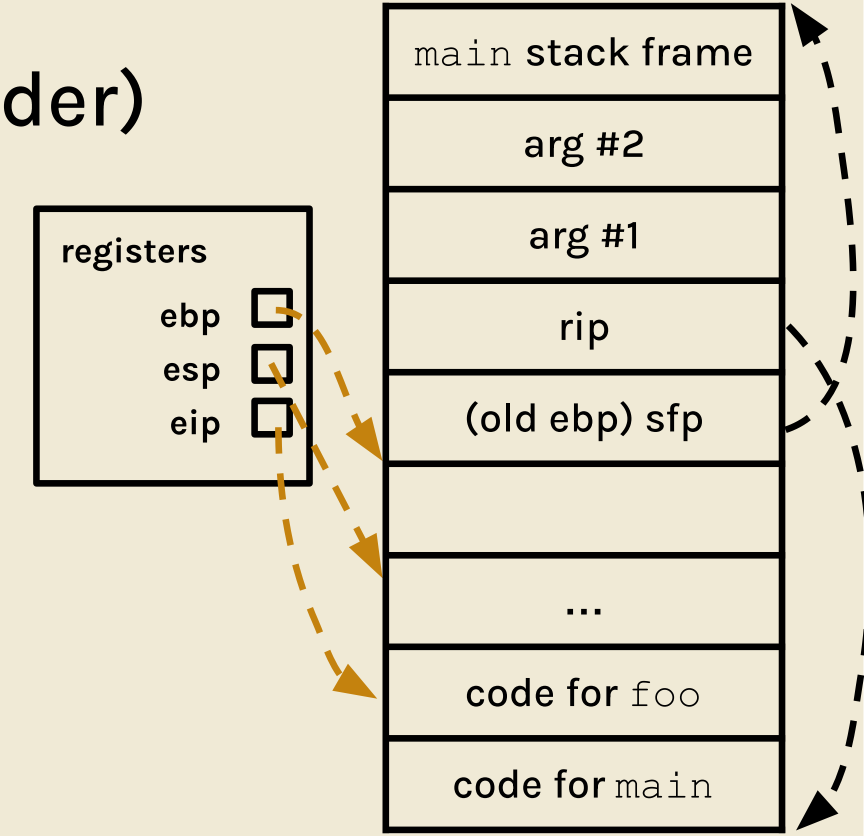
calling convention

1. push arguments (reverse order)
2. remember eip
3. remember ebp
4. adjust the stack frame
   - update ebp, esp, eip



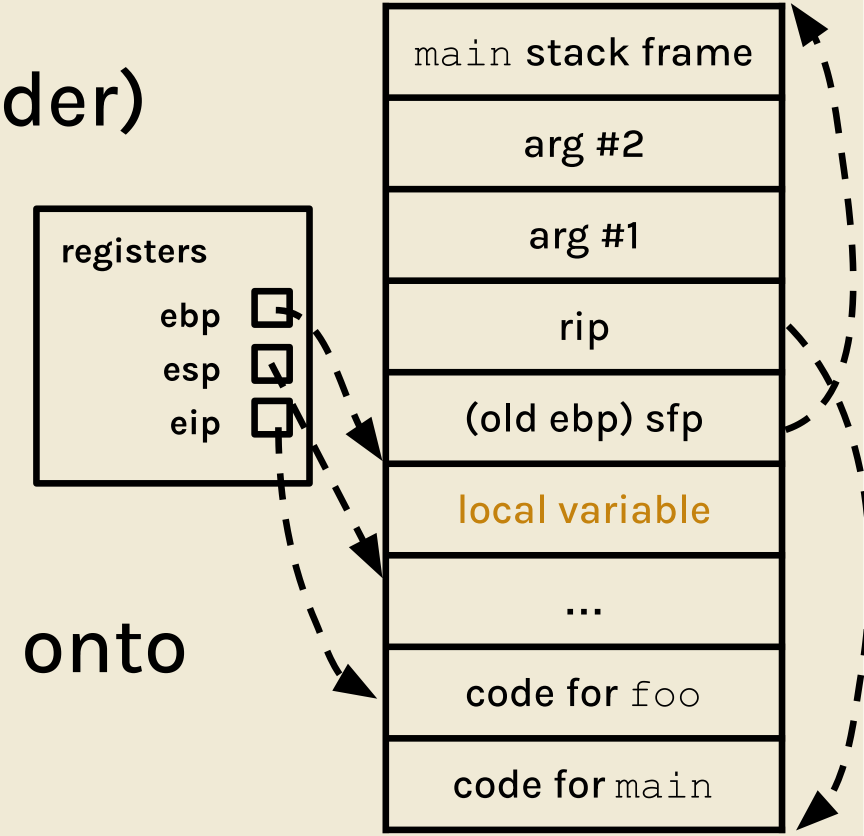
calling convention

1. push arguments (reverse order)
2. remember eip
3. remember ebp
4. adjust the stack frame
   - update ebp, esp, eip



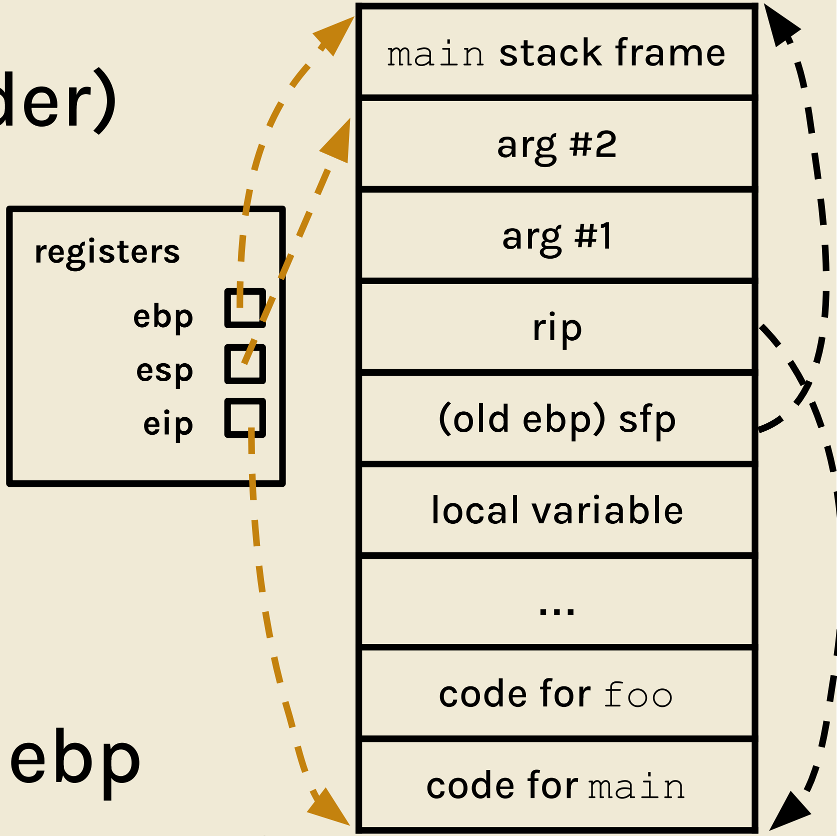
calling convention

1. push arguments (reverse order)
2. remember eip
3. remember ebp
4. adjust the stack frame
5. execute the function
   - and move local variables onto stack



calling convention

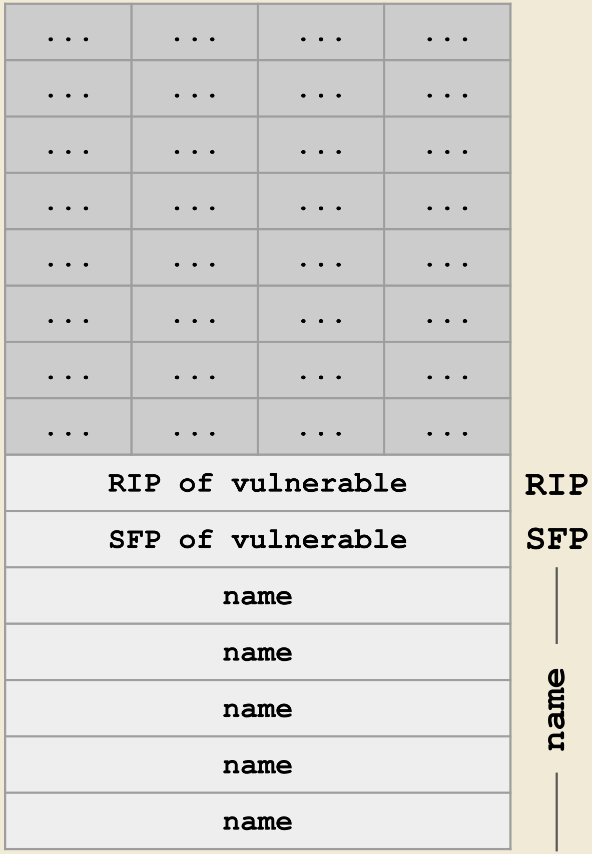
1. push arguments (reverse order)
2. remember eip
3. remember ebp
4. adjust the stack frame
5. execute the function
6. restore everything
   - use rip, sfp to restore eip, ebp
   - esp naturally moves up via popping



memory safety vulnerabilities

stack smashing,
signed/unsigned, etc.

overwriting the rip



... ...

... ...

... ...

... ...

... ...

... ...

... ...

... ...

RIP of vulnerable — RIP

SFP of vulnerable — SFP

name

name

name

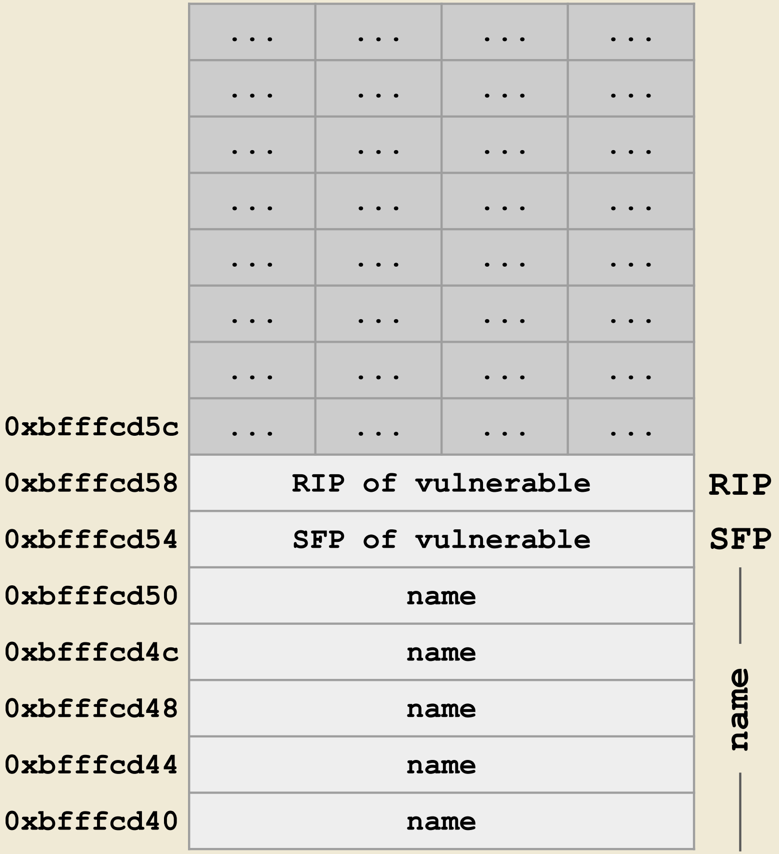
name

name

name

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

overwriting the rip



0xbfffcd5c ... ...

0xbfffcd58 RIP of vulnerable RIP

0xbfffcd54 SFP of vulnerable SFP

0xbfffcd50 name

0xbfffcd4c name

0xbfffcd48 name

0xbfffcd44 name

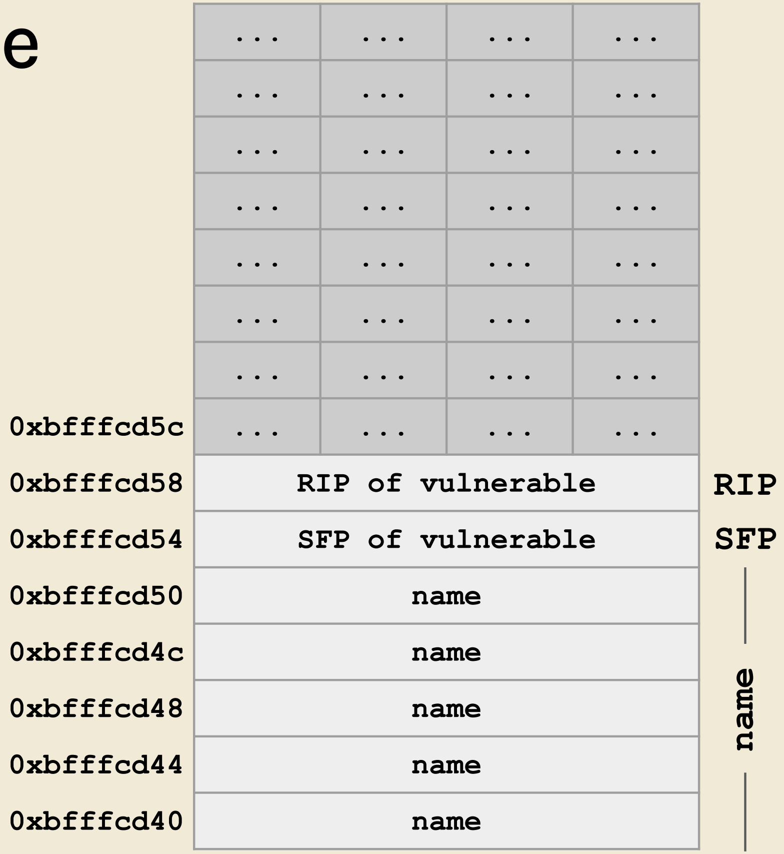
0xbfffcd40 name

name

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

overwriting the rip

- we have 12 bytes of shellcode

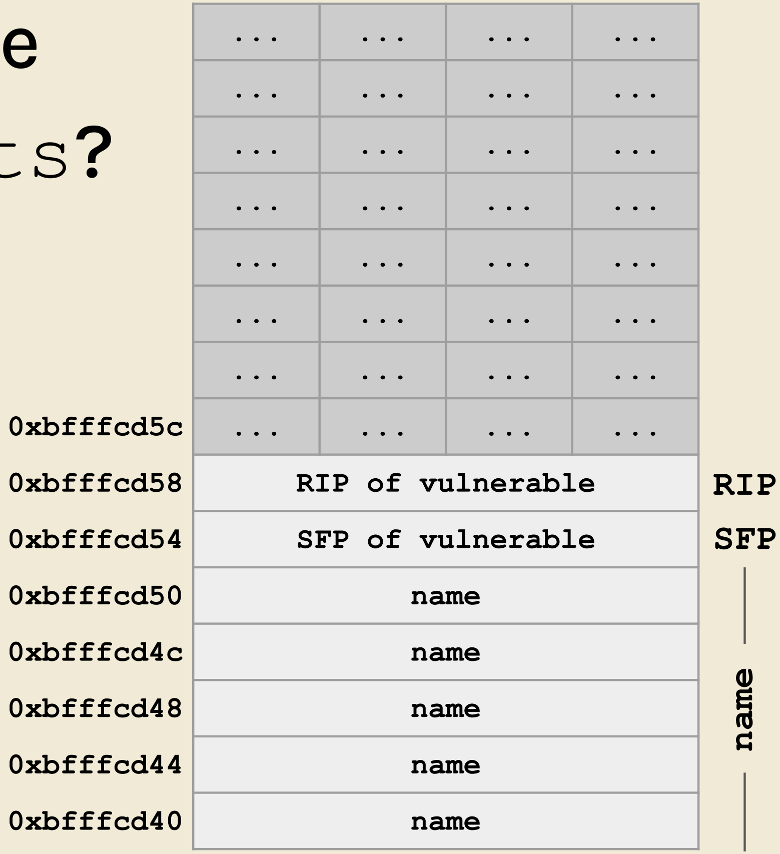


	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
0xbfffcd5c	...	...	...	...	
0xbfffcd58	RIP of vulnerable				RIP
0xbfffcd54	SFP of vulnerable				SFP
0xbfffcd50	name				name
0xbfffcd4c	name				
0xbfffcd48	name				
0xbfffcd44	name				
0xbfffcd40	name				

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

overwriting the rip

- we have 12 bytes of shellcode
- what input can I provide `gets`?



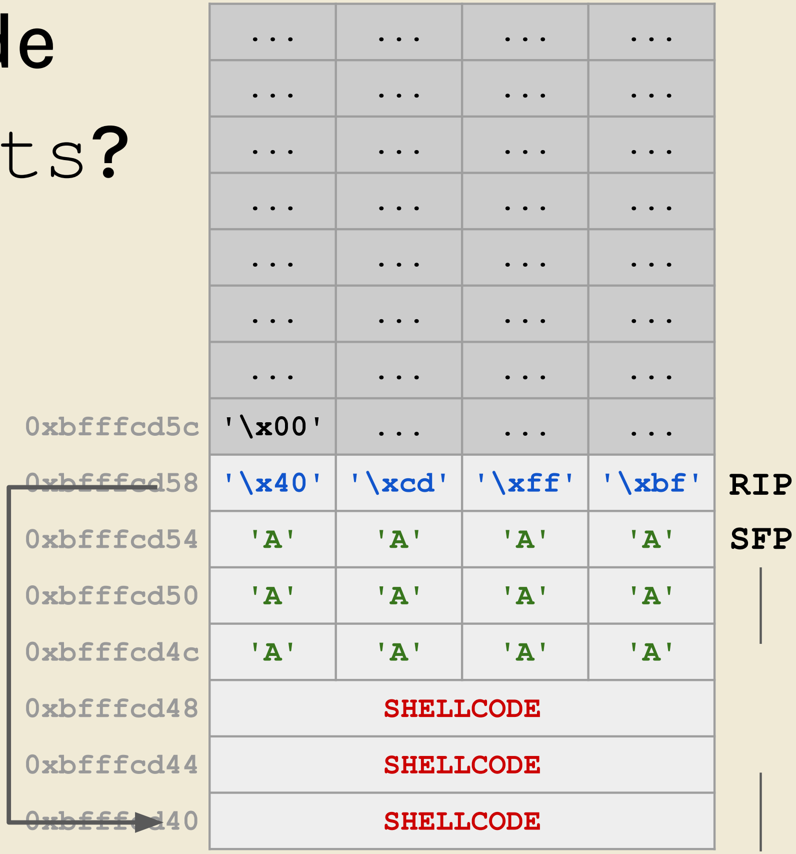
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
0xbfffcd5c	...	...	...	...	
0xbfffcd58	RIP of vulnerable				RIP
0xbfffcd54	SFP of vulnerable				SFP
0xbfffcd50	name				name
0xbfffcd4c	name				
0xbfffcd48	name				
0xbfffcd44	name				
0xbfffcd40	name				

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

overwriting the rip

- we have 12 bytes of shellcode
- what input can I provide `gets`?
- `SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
0xbfffcd5c	'\x00'	...	...	...	
0xbfffcd58	'\x40'	'\xcd'	'\xff'	'\xbf'	RIP
0xbfffcd54	'A'	'A'	'A'	'A'	SFP
0xbfffcd50	'A'	'A'	'A'	'A'	
0xbfffcd4c	'A'	'A'	'A'	'A'	
0xbfffcd48	SHELLCODE				
0xbfffcd44	SHELLCODE				
0xbfffcd40	SHELLCODE				

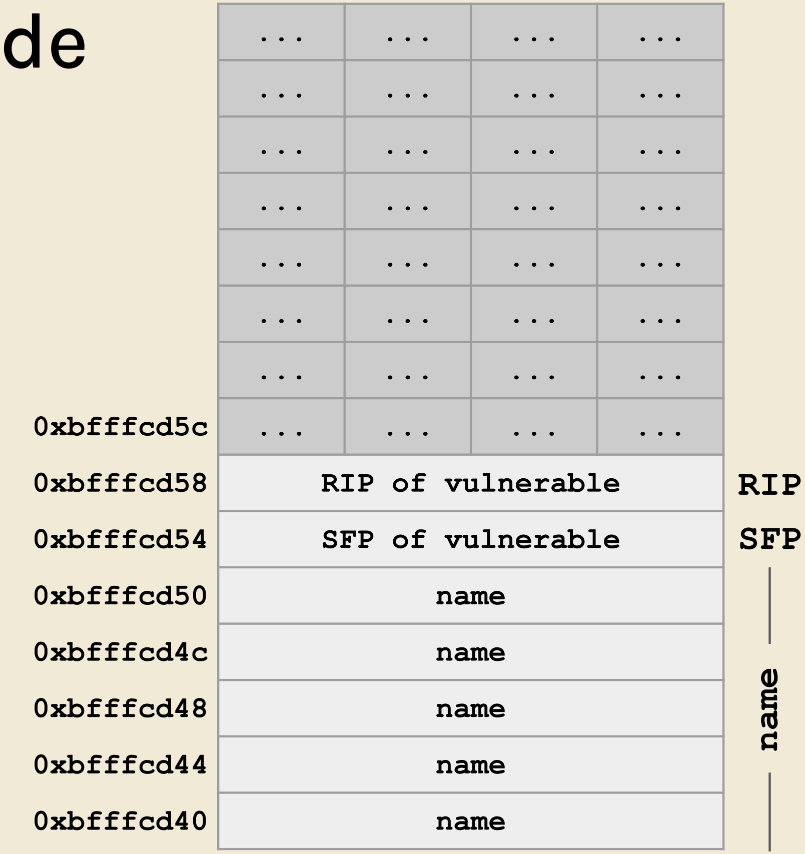
overwriting the rip

what if i have a very large shellcode?

overwriting the rip

- we have 28 bytes of shellcode

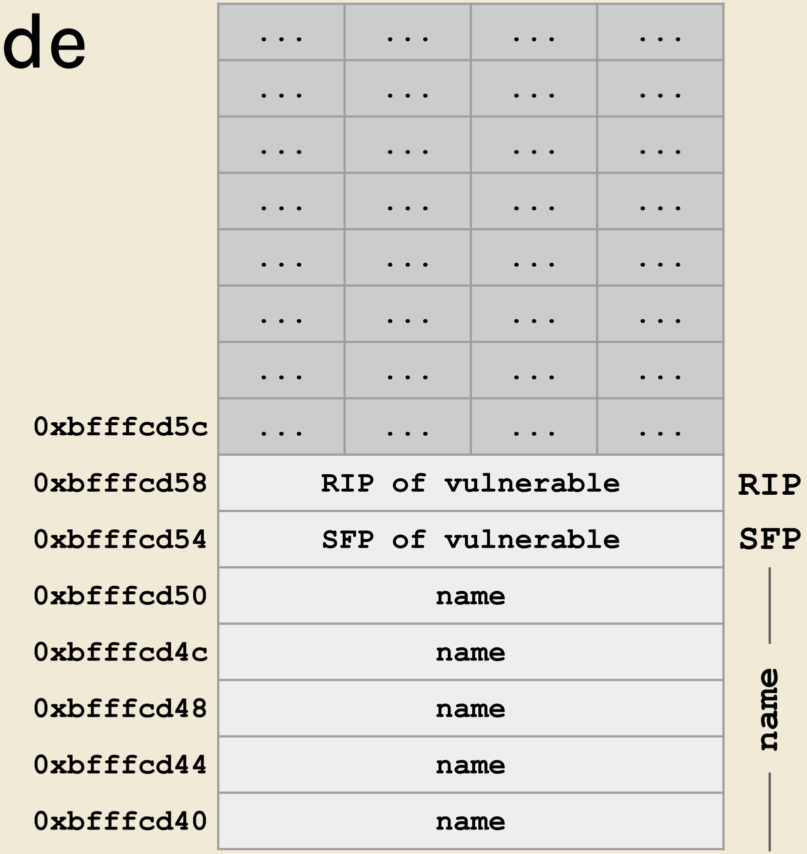
```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
0xbfffcd5c	...	...	...	...	
0xbfffcd58	RIP of vulnerable				RIP
0xbfffcd54	SFP of vulnerable				SFP
0xbfffcd50	name				name
0xbfffcd4c	name				
0xbfffcd48	name				
0xbfffcd44	name				
0xbfffcd40	name				

overwriting the rip

- we have 28 bytes of shellcode
- place shellcode AFTER rip



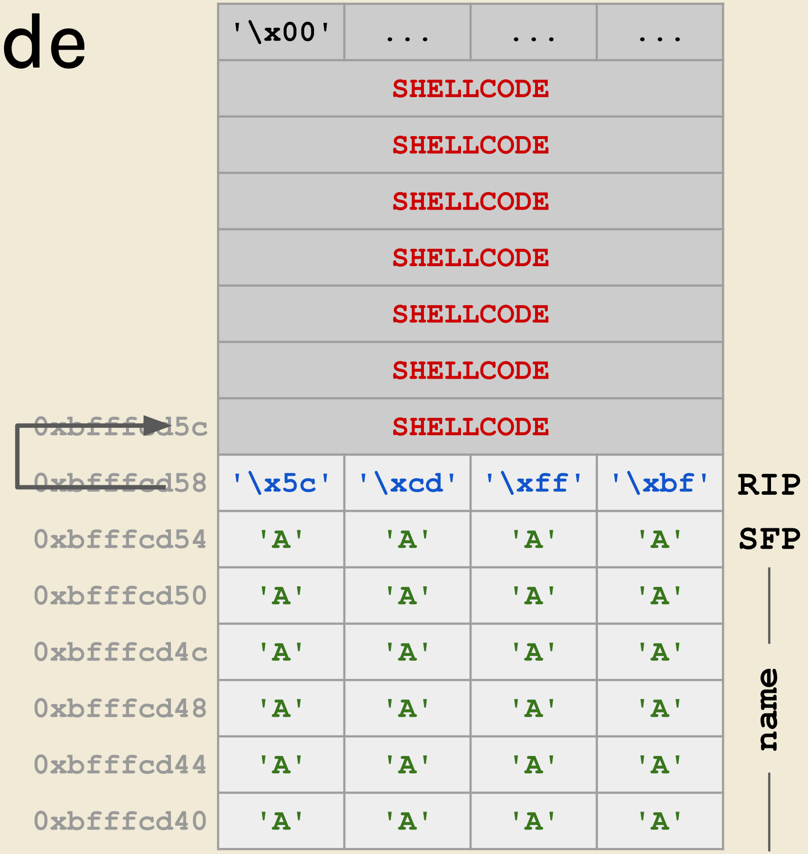
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
0xbfffcd5c	...	...	...	...	
0xbfffcd58	RIP of vulnerable				RIP
0xbfffcd54	SFP of vulnerable				SFP
0xbfffcd50	name				name
0xbfffcd4c	name				
0xbfffcd48	name				
0xbfffcd44	name				
0xbfffcd40	name				

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

overwriting the rip

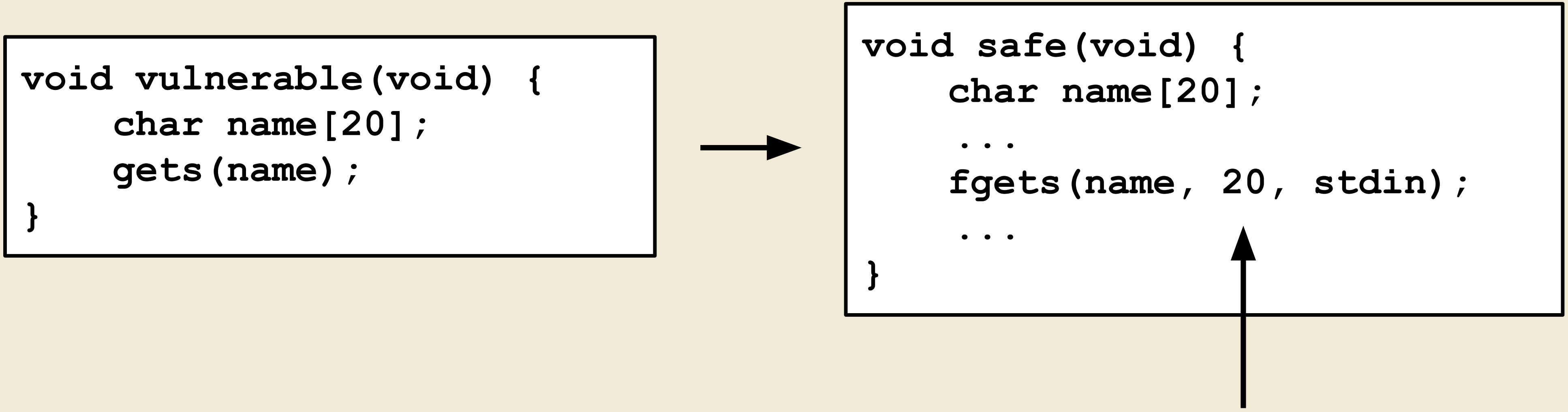
- we have 28 bytes of shellcode
- place shellcode AFTER rip
- 'A' * 24 + '\x5c\xcd\xff\xbf' + SHELLCODE

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



	'\x00'	...	...	...	
	SHELLCODE				
	SHELLCODE				
	SHELLCODE				
	SHELLCODE				
	SHELLCODE				
	SHELLCODE				
0xbfffcd5c	SHELLCODE				
0xbfffcd58	'\x5c'	'\xcd'	'\xff'	'\xbf'	RIP
0xbfffcd54	'A'	'A'	'A'	'A'	SFP
0xbfffcd50	'A'	'A'	'A'	'A'	name
0xbfffcd4c	'A'	'A'	'A'	'A'	
0xbfffcd48	'A'	'A'	'A'	'A'	
0xbfffcd44	'A'	'A'	'A'	'A'	
0xbfffcd40	'A'	'A'	'A'	'A'	

mitigating the `gets` vulnerability



```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

```
void safe(void) {
    char name[20];
    ...
    fgets(name, 20, stdin);
    ...
}
```

specify length!

unsafe C functions (not extensive)

- **gets** - read a string from stdin
  - use **fgets** instead
- **strcpy** - copy a string
  - use **strncpy** (more compatible, less safe) or **strlcpy** (less compatible, more safe) instead
- **strlen** - get the length of a string
  - use **strnlen** instead (or **memchr** if you really need compatible code)

signed/unsigned vulnerabilities

signed

```
void func(int len, char *data) {
    char buf[64];
    if (len > 64)      -1
        return;             -1
    memcpy(buf, data, len);
}
```

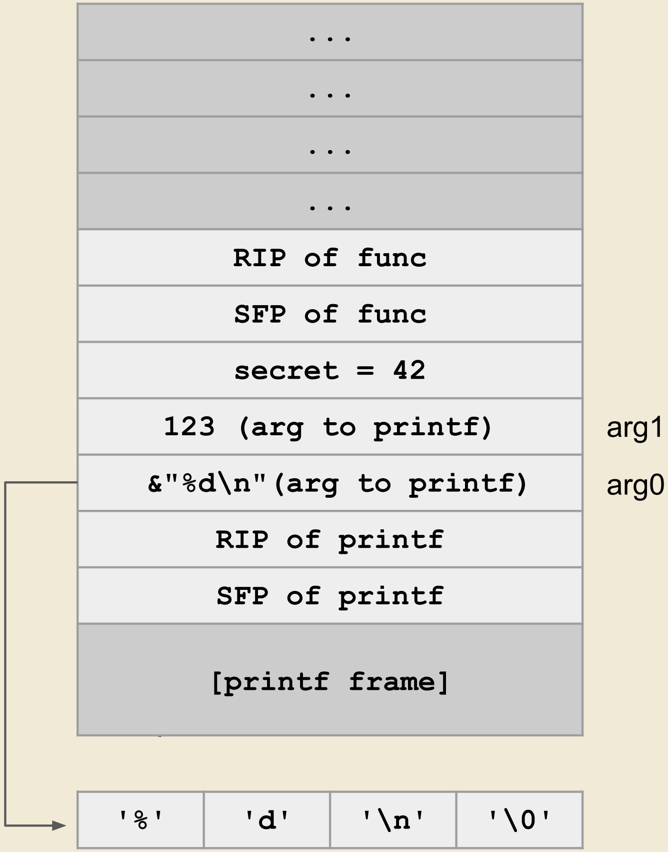
```
void *memcpy(void *dest, const void *src, size_t n);
```

unsigned

printf vulnerability

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

two arguments



...

...

...

...

RIP of func

SFP of func

secret = 42

123 (arg to printf) arg1

&"%d\n"(arg to printf) arg0

RIP of printf

SFP of printf

[printf frame]

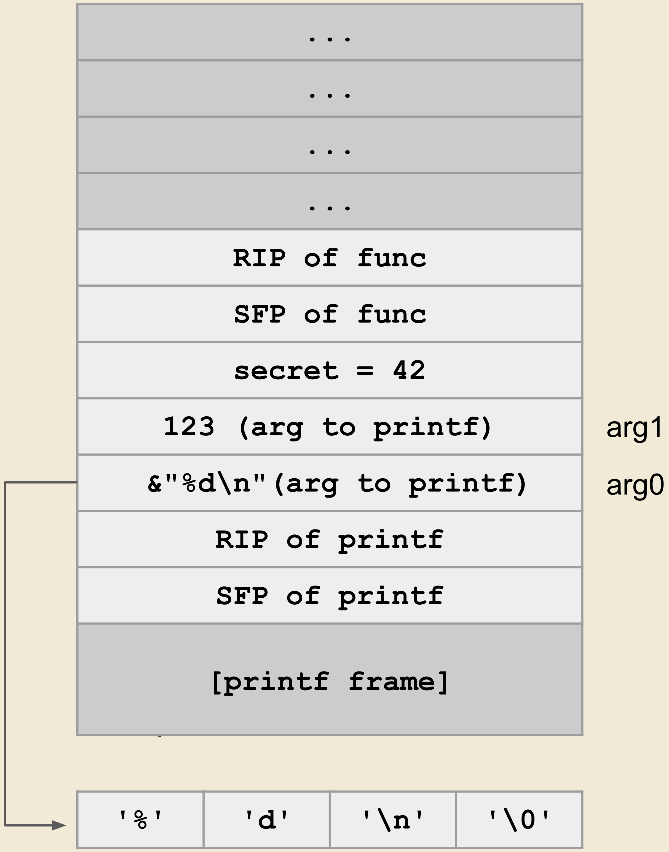
'%' 'd' '\n' '\0'

printf vulnerability

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

two arguments

what if there’s only one?



...

...

...

...

RIP of func

SFP of func

secret = 42

123 (arg to printf) arg1

&"%d\n"(arg to printf) arg0

RIP of printf

SFP of printf

[printf frame]

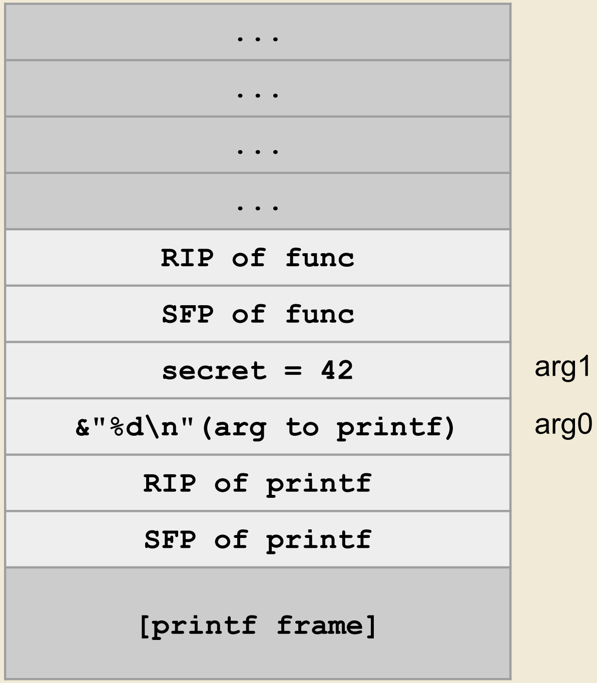
'%'	'd'	'\n'	'\0'

printf vulnerability

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

one argument

what if there's only one?



...

...

...

...

RIP of func

SFP of func

secret = 42 arg1

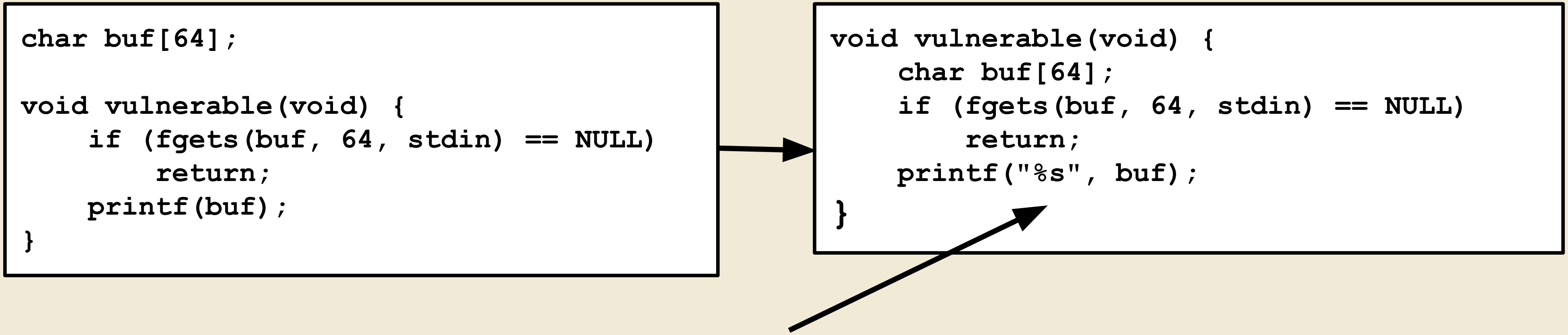
&"%d\n"(arg to printf) arg0

RIP of printf

SFP of printf

[printf frame]

mitigating printf vulnerabilities



```
char buf[64];

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

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

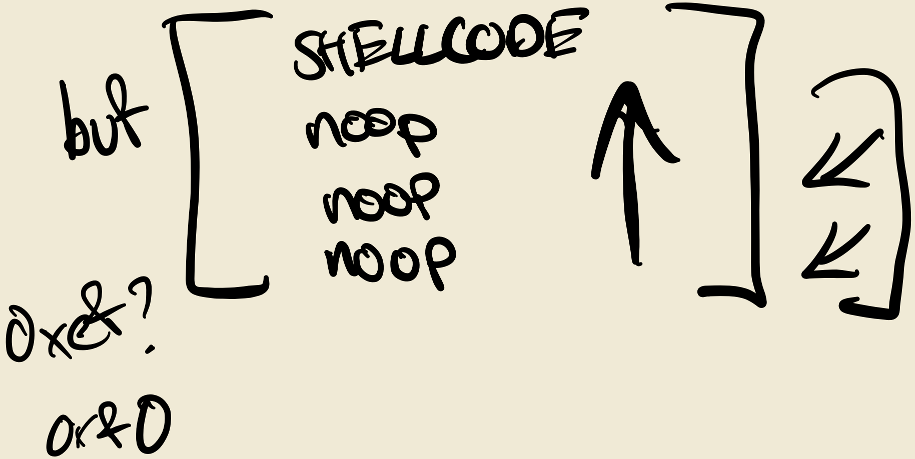
only accept trusted input!

heap vulnerabilities

- heap overflow
    - writes to buffers within objects in heap unchecked, can buffer overflow the vtable pointer of another object to point to shellcode
- use after free
    - free() memory, attacker writes to it, the freed object is used, which accesses malicious pointers

NOP sleds

- “no operation”, landing anywhere in a sequence of `noop`s (an x86 instruction) leads to your shellcode



worksheet

(on 161 website)

feedback

bit.ly/extended-feedback

slides: **bit.ly/cs161-disc**